GALiCA: A Gestural Approach to Live Coding Algorithms

by

Lark Savoldy

S.B., Computer Science and Electrical Engineering and Music,
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

© 2024 Lark Savoldy. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Lark Savoldy
Department of Electrical Engineering and Computer Science
December 13, 2023

Certified by: Ian Hattwick
Lecturer
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

GALiCA: A Gestural Approach to Live Coding Algorithms

by

Lark Savoldy

Submitted to the Department of Electrical Engineering and Computer Science
on December 13, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Live coding is an electronic music performance practice in which performers generate music and visuals in real time by writing code. The cognitive approach to live coding differs greatly from that of gestural music, in which performers leverage extensive embodied knowledge of their instrument. These two domains, which each provide unique tools for musical creativity and expressivity, are often performed separately.

This thesis considers the space between these two performance styles. The primary goal is to suggest the potential of a combined modality by considering techniques for gestural control over live code. A combination of live coding and gestural performance may allow for a new cognitive approach and entirely new ways to live code.

To explore this idea, this thesis introduces GALiCA, a live coding system that implements four techniques for manipulating code through gestural interaction with a MIDI controller. These techniques are facilitated by a flexible sequencer conceptualization that allows for easy modification. Additionally, to guide the analyses, this thesis synthesizes existing conceptual perspectives on the cognition involved in gestural performance and live coding. The promising results and analyses of these techniques may encourage further exploration into this new field and prompt new cognitive approaches to electronic music performance.

Thesis Supervisor: Ian Hattwick
Title: Lecturer

// Acknowledgments

I would like to thank my incredible friends, family, and colleagues, who have been a constant source of support and inspiration throughout my research. Their excitement about GALiCA has been instrumental in the development of this project. I would especially like to thank:

My advisor, Dr. Ian Hattwick, for providing invaluable guidance, enthusiasm, and feedback throughout my entire MEng program. Ian always made time in his busy schedule to excitedly discuss ideas with me, set up events for me to showcase GALiCA, and keep me on a timeline. None of this would have been possible without him.

Some of the other wonderful music faculty at MIT, Dr. Michael Scott Cuthbert, Dr. Emily Richmond Pollock, and Dr. Frederick Harris for fostering my love of music and encouraging me to pursue it in my studies and career.

My family, Carolyn, Louis, and Scott Savoldy, for always supporting my interests, celebrating my successes, and inspiring me by bravely pursuing their own passions.

My wonderful and brilliant friends, Julian Yocum, Margaret Shutts, and Torque Dandachi, for supporting me through many stressed nights and always motivating me with their dedication to the work that they do.

My roommates, Colin Greybosh and Eva Smerekanych, for listening to my roadblocks and giving me some much-needed human interaction every day.

And finally, GALiCA's biggest fan, Sanju Benjamin, for his overwhelming excitement about my work and never-ending support for both me and this project.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction **13**

1.1 Context . . . 14

1.2 Overview of GALiCA . . . 14

1.3 Thesis Structure . . . 16

1.4 Goals and Research Questions . . . 17

2 Background **19**

2.1 Electronic Music Performance . . . 19

2.1.1 Sequencers . . . 20

2.1.2 MIDI Controllers . . . 24

2.2 Live Coding . . . 27

2.2.1 Definition . . . 27

2.2.2 History and Performance Practices . . . 28

2.2.3 Why Live Coding is Unique . . . 28

2.2.4 Implications of Live Coding . . . 29

2.2.5 Examples . . . 30

2.2.6 Live Coding Sequencers . . . 31

2.3 Physical Embodiment . . . 31

2.3.1 Cognitive Background . . . 32

2.3.2 Cognitive Experience of Live Coding . . . 33

2.3.3 Previous Gestural Live Coding . . . 35

3 Conceptual Analysis **37**
3.1 Features of Music Performance . . . 37
3.1.1 Embodied Knowledge . . . 37
3.1.2 Coupling to Time/Real-Time feedback . . . 39
3.1.3 Precision and Accuracy . . . 40
3.1.4 Exploration . . . 40
3.1.5 Composition . . . 41
3.2 Combining Approaches . . . 41
3.2.1 Goals . . . 42
3.2.2 Challenges . . . 44

4 The Seq Class **47**
4.1 Sequencer Overview . . . 47
4.2 GALiCA Seq . . . 48
4.2.1 Overview . . . 49
4.3 Implementation Details . . . 51
4.3.1 Values and Durations . . . 51
4.3.2 Functions . . . 52
4.4 Parsing . . . 55
4.4.1 Recognizing Seq Instances . . . 56
4.4.2 Redefined Input Arrays . . . 56
4.5 Reflection . . . 57

5 MIDI Algorithm Interactions **59**
5.1 Overview of Implementations . . . 59
5.2 Ternary Expressions . . . 60
5.2.1 Goal . . . 60
5.2.2 Implementation . . . 61
5.2.3 Example . . . 64
5.2.4 Reflection . . . 68
5.3 Expression Variables . . . 70

5.3.1 Goal 70
5.3.2 Implementation 71
5.3.3 Example 73
5.3.4 Reflection 76
5.4 CC Callbacks and Variables 77
5.4.1 Goal 77
5.4.2 Implementation 78
5.4.3 Example 79
5.4.4 Reflection 80
5.5 MIDI Note Callbacks 80
5.5.1 Goal 81
5.5.2 Implementation 81
5.5.3 Example 81
5.5.4 Reflection 82

6 Conclusion **83**
6.1 Sequencer Design Analysis 83
6.2 Gestural Interaction Analysis 85
6.3 Contributions 87
6.4 Further Work 87
6.5 Conclusion 89

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

1-1 Overview of how GALiCA interacts with other systems. 15

2-1 Existing hardware sequencers. 23
2-2 Launchkey Mini 26
2-3 Common MIDI controllers 26
2-4 Human/Computer interaction feedback loop 34

3-1 Symbol/Sign/Signal interface analysis 42

4-1 A basic sequencer executing a step 48
4-2 Examples of different increment values stepping through an array . . 50
4-3 Example of sequencer output. 51
4-4 Flow diagram of GALiCA sequencer. 52

5-1 A dial selecting a value 65
5-2 A dial selecting an operator 65
5-3 The layout of the GALiCA Expression Variable interface 72
5-4 The Available Algorithms box, populated with user-added algorithms 73
5-5 The Algorithm Assignments box 74
5-6 A dial selecting available algorithms 75
5-7 Algorithm Assignments box after interaction 75

6-1 Flow diagram of the GALiCA sequencer. 85

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Live coding is an emerging musical performance practice in which a performer writes lines of code to generate sound real-time in front of an audience. Live coders often engage with their music through abstract concepts and symbols, thinking of ways to convert ideas into code. This performance practice frees the live coder from handling every moment's sonic output, allowing the live coder high mental bandwidth. This gives a live coder more freedom to consider high-level ideas than an acoustic instrumentalist might have.

While this feature of live coding is very powerful, there are other music performance practices that use real-time gestural interaction to great effect. Gestural musicians can form embodied knowledge about their musical system, which is cognition that occurs outside of direct conscious decision-making. For example, a guitarist knows how to strum an A chord without having to consider every finger's position: they only need to think of forming an A chord, and their fingers will fall into place. This is a very effective and expressive performance technique. Although live coding uses embodied knowledge to some extent, as it requires the transduction of an idea into code through a keyboard, it does not rely on gestural interaction enough to create strong embodied playing modes.

The goal of this thesis is to examine the potential in combining these two seemingly distinct performance modalities: live coding and gestural performance. If the two can be integrated in a way that preserves the unique performative benefits of both, then

live coders may be able to find new ways of performance and greater creative potential in live coding systems.

1.1 Context

Live coding is an extension of a much broader electronic music performance practice. There are many modes of interaction with these live systems, ranging from gestural electronic systems that are played similarly to a traditional acoustic instrument, to music systems that generate experimental sounds through complex and undecipherable algorithms. A common theme of this field is finding novel interaction techniques that allow for different modes of expression. This thesis aims to explore new intersections between these existing practices and show the potential creative possibilities of a system that incorporates both code-based and gestural interactions.

Other elements of electronic music performance are important in the development of this thesis: MIDI controllers and sequencers. In order to create a system that explores gestural live coding, there must be a way of transducing gesture into software signals, and a way of using the code to produce sonic output. As illustrated in figure 1-1, this thesis uses the MIDI protocol to receive data from gestural controllers, and also to transmit control data to software synthesizers. This thesis also builds on the concept of a sequencer, a common means of scheduling sonic events in electronic music performance, to generate messages for controlling sound synthesis. Both of these techniques are very prevalent in electronic music and draw on existing practices.

1.2 Overview of GALiCA

In order to explore gestural interaction with algorithms, this thesis introduces a new live coding system called GALiCA, with the the following features:

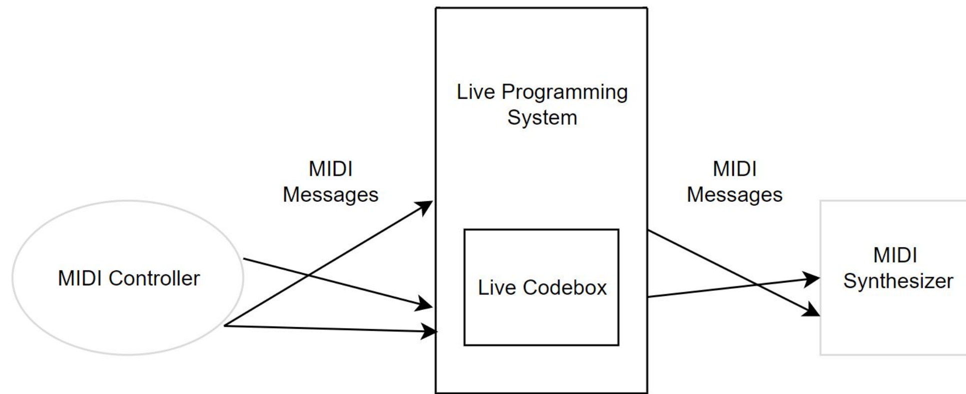


Figure 1-1: The GALiCA interface acts as a middle layer, receiving MIDI input from a controller, using this input to modify code written in the codebox, and sending MIDI messages to a user-chosen synthesizer.

- Uses a MIDI controller as primary input: although the system works with only a codebox, its primary focus is implementing intuitive methods to control user code using a MIDI controller.

- Receives MIDI input and sends MIDI output: The system essentially acts as a middle layer which receives MIDI input from a controller, uses the MIDI input to modify user-written code, then outputs MIDI messages to a user-chosen synthesizer.

- Runs in the browser and employs JavaScript for user code input: the system requires no local downloads and uses a language which many programmers are already familiar with in order to decrease the barrier to entry.

- Creates timed musical events: the system employs a sequencer implementation that can generate timed musical events. The user can create and interact with instantiations of the sequencer through both the codebox and the MIDI controller.

- Uses user-defined MIDI mappings: the system allows users to map incoming MIDI messages to features in the code. The user can choose how interacting the MIDI controller will affect the code they have written.

- Implements gestural algorithmic manipulation: the system supports four different approaches for interacting with live coded algorithms through MIDI controller input.

The source code for this system can be found at: `https://github.com/hsavoldy/GALiCA` [46].

1.3 Thesis Structure

The primary contribution of GALiCA is the implementation and analysis of four gestural approaches to live coding. An important supporting contribution is the system's sequencer concept, which is integral to the development of these approaches. All of the work done in this project is motivated by a discussion of conceptual perspectives on gesture, live coding, embodiment of knowledge, and sequencers. To explore these ideas, this thesis is split into five main sections:

- Chapter 2, the Background, provides an overview of current research relating to electronic music practices (such as sequencers), live coding, and cognitive theories about embodiment. Its contents provide the basis for many of the ideas presented in this thesis, specifically the conceptual analysis that motivates the purpose of this project.

- Chapter 3, the Conceptual Analysis, analyzes previous research to present the possible implications of incorporating gestural interaction into a live coding system. It explores the differences and commonalities of these two approaches and considers the effectiveness of a combined modality.

- Chapter 4, the Seq Class, introduces the sequencer conceptualization used in this thesis. It details a conceptual overview and includes a discussion of the Seq class' implementation in the code.

- Chapter 5, MIDI Algorithm Interactions, describes the work that addresses the primary objective of this thesis. It explores four new approaches to using

gesture to modify live coded algorithms. It broadly describes each approach, how the approach is implemented in the code, and a reflection of the usefulness and potential of each approach.

- Chapter 6, the Conclusion, summarizes the contributions and analyses of this thesis. It also considers the future work that would further advance this field.

1.4 Goals and Research Questions

The main goal of this project is to explore the creative potential of integrating live coding, a predominantly symbol-level practice, with the use of a hardware gestural controller. Although live coding systems with support for MIDI controllers exist, they do not generally view the hardware as a means of manipulating the code algorithmically. As such, there is not a good understanding of how physical controllers can be used to modify code, and more specifically, how they can be used to turn high-level algorithmic concepts into embodied knowledge which can be drawn from in a live performance.

To facilitate this exploration, the design of the GALiCA system is a primary concern as well. Particularly, GALiCA's central interaction mode, a sequencer, must be designed to integrate gestural control as intuitively as possible. Because of this, creating a flexible and useful sequencer concept is another main goal.

To guide the development and analysis of these goals, three primary research questions are investigated in this thesis:

- How can live coding effectively incorporate gestural interaction? How could this approach recruit the cognitive benefits of gestural performance, such as the physical embodiment of knowledge, to increase the creative potential of live coding?

- How can a standard MIDI controller facilitate gestural engagement with a symbolic code editor?

- What sequencer design characteristics enable users to leverage both gestural and symbolic interaction to maximize expressive potential?

Chapter 2

Background

GALiCA was created within the context of electronic music performance practice, specifically the burgeoning practice of live coding. Other relevant topics within this practice are the concept of a sequencer, as it exists within the history of electronic music performance and as it is used today, as well as MIDI controllers. The first two parts of this section will discuss the relevant background of these areas.

Another core aspect of this thesis is exploring how physical knowledge is embodied, specifically within a music performance system. The last section provides a brief discussion of the current literature establishing theories of embodied knowledge and how they relate to musical performance, including live coding.

2.1 Electronic Music Performance

Electronic music performance is a practice that comprises a diverse set of performance and aesthetic processes, but it is often characterized by novel uses of contemporary technology, an emphasis on live interaction with electronic interfaces, and the use of electronic mediums as a way to push the boundaries of music. As Collins [13] states, "since the 1930s (well before the advent of tape) composers have been using this property of electronics to produce not just new sounds but fundamentally new approaches to organising the sonic world." Historically, a core component of live electronic music is questioning mainstream ideas about music performance, resulting in

often experimental performances that take place in niche underground venues and bars rather than large concerts [8]. It is through this lens that this thesis considers sequencers: not as tools to generate pre-produced or mainstream music, but as methods for performers to engage with sound live in front of an audience.

The predominant sequencer concept, which is used in both hardware and software music performance systems, is based off of historic hardware sequencers. Analog sequencers first emerged in electronic live music around the 1960s, with the popularization of modular synthesizers such as the Moog [31] and the Buchla series [10]. The sequencer module was used to schedule signals, which could be routed to other modules. Often these signals were used as note values. This gave performers the ability to create intricate and real-time modifiable melodic lines, which once started, would continue playing without any oversight from the performer. Because of the usefulness of this module, sequencers became ubiquitous in modular-synthesizer racks, and many electronic musicians became familiar with the mental model of a sequencer [31]. Years later, as live coding emerged, many systems adopted the sequencer concept as a way for the live coder to create sonic events. [1]

2.1.1 Sequencers

A sequencer is a means of scheduling musical events in time. Dannenberg [16] suggests that "music is the presentation of sound in some form of temporal organization." Viewed through this perspective, sequencers create sound events that are precisely temporally defined within this organization. Because of the broadness of this definition, sequencers exist in many different forms and for many different applications.

Existing Sequencer Taxonomies

There are a few existing approaches to defining and classifying sequencers. Duignan et al. created a taxonomy of sequencer user-interfaces, introducing criteria such as abstraction level, linearization stage, and event ordering [20]. This taxonomy ap-

[1] For a full overview of the history of electronic music performance practices, read Joel Chadabe's *Electric Sound* [11].

plies broadly to all sequencers, but this thesis is concerned with the application of sequencers in live performance, and as such will limit the discussion to sequencers which can be modified in real-time. In the terms of Duignan et al., these sequencers have eager linearization and control event ordering, although the level of abstraction can vary.

In another view on performance using sequencers, Püst et al. have introduced a taxonomy of interactions in electronic music performance [44]. This taxonomy is applied to two primary modes, that of sound design and sequencing. They argue that performers often alternate between these two modes of performance, and that there are four interaction classes which cut across these modes: creation, modification, selection, and evaluation. They continue to use these interactions to evaluate a range of hardware including sequencers and synthesizers.

This paper highlights the prominence of the sequencer as a primary performance tool, and the interaction classes provide a framework to reflect on what occurs in the act of performance. These interaction classes may be seen across a wide variety of performance techniques, including gestural and live coding practices, but the way they manifest may be different depending on the context.

A Taxonomy of Hardware Sequencers

The previously mentioned taxonomies are useful considerations and are used to motivate a taxonomy of interaction types that are considered in the development of GALiCA. In table 2.1, a taxonomy is applied to five popular hardware sequencers with different algorithmic approaches: Doepfer's A155 [18], intellijel's Metropolis [26], Make Noise's René [41], Buchla's 250e Dual Arbitrary Function Generator [12], and the Klee sequencer [22]. The goal of this analysis is to uncover what interaction modes are possible during live performance with a sequencer. This taxonomy considers the following characteristics:

1. Algorithmic Approach: what approach guides the use of the sequencer. Each sequencer can perform basic sequencing, but beyond that, they each have a

Sequencer Example	Features			
	Algorithmic Approach	**Step Increment Manipulation**	**Step Index Manipulation**	**Variable Sequence Length**
A155	Basic		Reset	
Metropolis	Gates per step	1 to 8 pulse counts per step	Forward, Reverse, PingPong, Random, Brownian	Ability to skip steps
René	Cartesian		Snake, Cartesian, Presets	Toggle each step
Buchla 250e	Circular	Dial-adjustable time values for each step	Stage addressing input	Nested Loops
Klee	Shift/bit registers		Random, Pattern	8 or 16 steps

Table 2.1: Feature comparison of different hardware sequencers

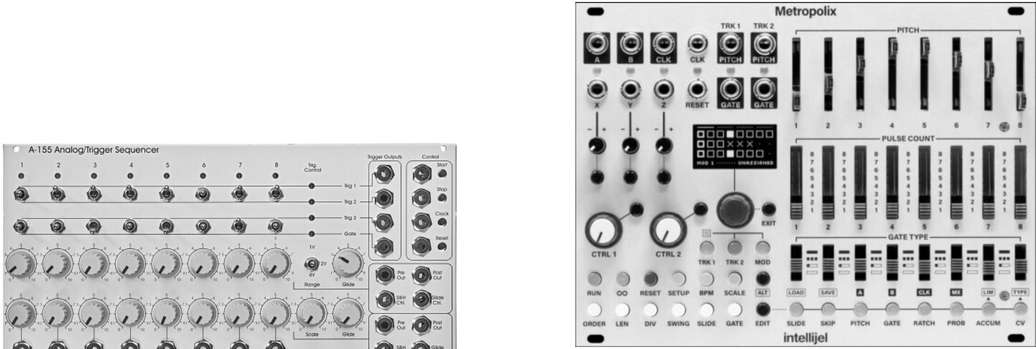
novel concept that informs how they are performed. For example, the René operates in a 2D Cartesian space rather than a 1D sequence. The Klee, on the other hand, uses a 1D sequence but sums the voltages of multiple steps based on a user-defined bit pattern.

2. Step Increment Manipulation: a sequencer's ability to change its step increment value, in other words, how the duration of the step can be altered. For example, the A155 lacks the ability to perform a step that is any different length than the clocking rate, so it cannot manipulate the step increment.

3. Step Index Manipulation: a sequencer's ability to perform a non-linear pattern, by altering the current index of the sequence. For example, the Metropolis includes the ability to execute the sequence in reverse or in a PingPong pattern. The A155 can technically alter the step index by allowing a reset input, which will return the sequence to index 0.

4. Variable Sequence Length: a sequencer's ability to change the length of the sequence. The A155 cannot modify how long the 8 step sequence is (besides by

manual input to the reset input), while the Buchla 250e very clearly integrates creating a sequence that is some fraction of the entire circle of steps.



(a) The Doepfer A155 [19]

(b) The Intellijel Metropolix [27] (a new version of the Metropolis)

(c) The Make Noise Co. René [36]

(d) The Buchla 250e [9]

(e) Scott Stites' Klee Sequencer [21]

Figure 2-1: The interfaces of the five examined sequencers.

Taxonomy Discussion

The design of these sequencers all suggest very different user interactions. For example, the A155 and Metropolis display a linear sequence, the Buckla 250e uses a circular sequence, and the Rene uses a Cartesian, grid-like sequence. They all differ

in interaction modes: the A155 clearly displays all actions through physical, single-mapped dials and switches, while the Buchla 250e uses menus to obscure interactions that the user must select. Some employ saved sequences, enabling performers to call on previously created material, while others are maintain no memory and irrevocably change throughout a performance but ensure that the state of the sequencer interface always matches the state of the internals. Some interactions are relatively obscure in their sonic outcomes, such as the consequences of shifting a bit in the Klee, while interactions with the A155 are easily mentally tracked. All these sequencers have different goals and purposes. The Buchla 250e is not even marketed as a sequencer, with the manufacturer instead opting to call it a "function generator".

Despite their differences, all of these sequencers enable a performer to engage real-time with a sequence through a chosen set of interactions. Each listed sequencer could do what a A155 can do: execute a linear series of steps, with one step occurring on each beat, whose value is chosen by an analog dial. However, the additional interaction modes shift the way that a performer conceives of a performance. The sequences that can be produced by a Cartesian layout may sound identical to those produced by another means, but a performer using a René will have to implement any sequence through a Cartesian conceptualization of a sequencer.

In live coding, it is possible to create a sequencer that can do everything these sequencers can do, as well as boundlessly more. However, the sequencer model must be presented in the code in some way. This impacts the performer, just as presentation informs interaction modes for hardware sequencers.

2.1.2 MIDI Controllers

A primary method through which electronic music performers gesturally engage with their systems is through MIDI controllers. MIDI controllers are physical interfaces that send messages about note events and parameter values to a chosen channel using the MIDI protocol.

MIDI

MIDI (Musical Instrument Digital Interface) is a message protocol designed for digital music devices to communicate with each other. One of the primary reasons MIDI was created was to allow musicians to gesturally control electronic music systems. MIDI controllers can take many different form factors, such as keyboards, drum pads, or DJ controllers. MIDI messages can be broadly organized into note events (note on and note off), continuous controller (CC) events (sending a value from 0-127), and other system-wide messages, such as reset or clock messages. [34]

1. Note events: There are two note events, note on and note off. A note on event includes information about the pitch of the note, the channel, and the velocity. MIDI note pitches assign integer values to notes within a 12-tone equal temperament system, where a C4 is a MIDI note 60. The velocity of the message corresponds to the volume of the note, and the channel indicates which channel the MIDI note should be played on. Note off events indicate that the note sent by a previous note on event should stop. It typically includes a MIDI note pitch and a channel number.

2. CC messages: These messages include a value from 0-127, a controller number, and a channel number. The controller number indicates which controller of the system should be adjusted to the CC value. There are a set of standardized CC controller number mappings, such as controller number 7 corresponding with volume control, although they are less well specified than MIDI notes [29].

3. Clock: Among the system-wide events, MIDI supports a MIDI clock. This is a simple message that sends at a rate of 24 ppqn (pulses per quarter note) based on the tempo of the system. This allows multiple MIDI systems to stay in sync with each other, with one system sending MIDI clock pulses, and the others receiving them.

4. Other system-wide events: MIDI contains a variety of other message types, such as reset, start, stop, and song position pointers.

Controller Examples

While all MIDI controllers communicate through the standardized MIDI protocol, the controllers themselves can take any form factor. Many MIDI controllers mimic existing instruments, most commonly keyboards. The Launchkey Mini [42] is an example of a MIDI keyboard with both keys that send MIDI notes as well as dials and pads which send CC events.

Figure 2-2: MIDI Keyboard Example, the Launchkey Mini (photo by Lark Savoldy)

MIDI controllers that mimic many other instruments exist as well, including drum kits, wind instruments, and guitars.

Other common MIDI controllers include pad controllers, with an array of sensitive pads which send either note or CC messages, and DJ controllers, with dials, faders, and spinning discs which typically send CC messages.

(a) The Novation Launchpad S (photo by LeoKliesen00 [33])

(b) The Vestax VCI-380 (photo by TSchenke [50])

Figure 2-3: Common MIDI controllers

2.2 Live Coding

Live coding is an electronic music performance practice in which a performer, or live coder, uses a programming language to generate sound in front of an audience. It comes from a rich history in electronic music performance and, because it is often used as an experimental art form, there are many discussions surrounding what live coding is and how it operates as a performance practice. For example, *Live Coding: a User's Manual* [8] considers "how the performance of live coding proposes new ways of operating, posing questions and challenges to some of the underpinning values and ideologies of a wider computational culture." These ideas motivate the research done in this section.

2.2.1 Definition

Defining live coding is an open question among live coders. For example, *Live Coding: a User's Manual* [8] states that "live coding has been described in terms such as writing software in real time, changing a program while it is running, projecting the screen for the audience to participate in, writing as an improvisatory practice, composing live using textual notation, changing rules while following them, conversational programming (conversing with the computer in its own native language), thinking in public, and creating and using bespoke systems tailored for on-the-fly or just-in-time performance." Live coding is a continuation of historic electronic music performance practices, using general-purpose computers to perform rather than specialized hardware. As a part of this tradition, live coding aims to push the boundaries of live performance. Therefore, "live coding" is not a strictly defined concept; some even claim that it should defy definition by its nature: David Ogborn [8] states, “To define something is to stake a claim to its future, to make a claim about what it should be or become. This makes me hesitate to define live coding.” Even TOPLAP, an organization created to explore and promote live coding, released a manifesto on computer music performance that has been only a draft since its creation in 2004 [49].

2.2.2 History and Performance Practices

Live coding emerged when personal computers became prevalent enough and computationally powerful enough to be useful as an audio synthesis tool. This practice took hold in the 1990s: live coders would bring their laptops to clubs and bars to perform. Often, the audience would watch with no additional visual information, but eventually, common practice shifted to projecting the code being written. As the TOPLAP manifesto draft [49] states: "Show us your screens."

During this time, performers would use programming systems with an emphasis on real-time audio processing, such as SuperCollider [38], to live code. Eventually, languages created specifically for live coding were released, such as the ChucK on-the-fly language by Ge Wang [51]. Many smaller, localized movements of live coding began to converge in meetings and workshops, and TOPLAP formed, giving live coding a name and a basis as a live performance technique.

Interest in live coding continued to spread, culminating in the creation of dozens of live coding languages [48]. Today, live coding takes many forms. Festivals for live coding, called Algoraves, feature live coders creating music for the audience to dance to [17]. Live coding is also a subject of academic interest: an international conference dedicated to live coding, ICLC (the International Conference on Live Coding) began 2015 [28], and NIME (The International Conference on New Interfaces for Musical Expression), a premier conference for live electronic performance research, has featured 15 publications on live coding since 2007 [40].

2.2.3 Why Live Coding is Unique

The computer music that predated live coding shared many similarities with it, relating to the innumerable possibilities of an instrument that can be coded. The League of Automatic Music Composers was an influential music group who pioneered computer music, forming in 1977 [7]. Members of this group, John Bischoff and Tim Perkis, created the album Artificial Horizon in 1988 [6] and wrote on the CD sleeve: "for us, composing a piece of music is like building a new instrument, an instrument

whose behaviour makes up the performance. We act at once as performer, composer and instrument builder, in some ways working more like sculptors than traditional musicians." This is the reality of live coding as well. Live coding differs considerably from other live music practices (electronic and acoustic). It allows the performer immense freedom: any idea that a live coder can translate into code can be conceivably created during a performance. A live coder can choose to tweak small parameters in the code, create new structures that drastically change the sound, even step back and listen to how the sound evolves without their constant participation. Live coding is a completely new way to engage with music and algorithms.

2.2.4 Implications of Live Coding

Because of the unique elements of live coding performance, there are many ways that differences in performance may manifest. Nick Collins explores these ideas in a paper he published in 2003 [14]. Some implications he points out are:

1. Arbitrarily complex changes in structure at performance time at the expense of the high risk of running code: Although a live coder can create anything and structure the sound however they can imagine, a live coder cannot debug real-time. Any unintended bugs in the code will become present in the performance the moment they are run. This puts a limitation on novel exploration during a performance, because trying new material may result in unwelcome sonic outcomes.

2. Computer languages are immensely rich, infinite grammars, but typing is not a visually interesting performance method: using a programming language as the mode of performance is incredibly powerful, as it allows the performer to create anything. However, the audience may not be engaged by watching a live coder bent over a computer as much as they may be in other kind of instrument performance where a musician's clear physical movement translate to sound.

3. Live coding creates a great intellectual challenge, but the concentration required

diminishes with the stress of performance: live coding requires a high level of concentration because an incorrect keystroke can cause an error.

2.2.5 Examples

Live coding systems vary across many axes: many use a sequencer as a primary interaction, some are built off of existing programming languages, and some do not even use a traditional codebox, such as ORCΛ [45]. Some systems primarily use samples to generate sound while others use oscillators and filters, calling to concepts used in modular synthesizer racks. Four examples of popular yet distinct live coding languages are explored below.

Gibber

Gibber [23] is a web-based live coding environment that uses JavaScript syntax. Sonic events are sequenced and can take the form of samples or modular synthesizer-style sound generation and modification.

TidalCycles

TidalCycles [39] is a system that uses SuperCollider as its audio engine, and Haskell syntax for codebox input. Tidal also sequences sonic events and primarily uses audio samples as its sonic output, although it is also capable of synthesizing sounds through SuperCollider.

Sonic Pi

Sonic Pi [1] is a stand-alone application that uses Ruby for its syntax. It employs a loop-based approach to sonic event creation and also has strong support for samples as well as synthesized sounds

2.2.6 Live Coding Sequencers

A commonality among live coding languages is the ability to sequence musical events. However, these sequencers are implemented in different ways, drawing on different sequencer conceptualizations. Tidal Cycles, Gibber, and Sonic Pi all conceptualize sequencers in different ways.

- In Tidal Cycles, a concept of a sequencer is broadened into the term "cycle." A series of musical events, such as drum hits or notes, can be written to occupy one cycle, splitting the time of the cycle evenly among them by default. This is a powerful tool, because it allows the live coder to create musical phrases that always fit into one cycle, ensuring rhythmic consistency across different structures in the code.

- In Gibber, the Seq class is the primary method of creating musical events. A Gibber Seq object is defined with an array of values and an array of durations, which operate separately, unlike a Tidal Cycle. Gibber's philosophy is that anything can be sequenced, consistent with traditional hardware sequencers, which can sequence any arbitrary values.

- SonicPi does not have an explicit data structure that corresponds with a sequencer. Instead, musical events are placed inside loops which are called at specified rates. In practice, this loop-based approach functions very similarly to a sequencer.

Despite these syntactic differences, all systems can be used to create the same sequences. However, the way in which a live coder would create that same sequence could vary drastically across these three systems.

2.3 Physical Embodiment

Playing an instrument is inextricably tied to physical movement: an instrumentalist must transduce a mental process into physical movement in order to produce a sonic

response. The physicality of playing an instrument is an important factor in how it is played and conceptualized. This is true for live coding practice as well: typing on a physical keyboard is an embodied act. However, the physical embodiment of music in a gestural performance can be quite different from that of a live coding performance. This section will explore these similarities and differences after establishing the current psychological theories of physical embodiment.

2.3.1 Cognitive Background

The neural processing of music is strongly linked to physicality. Janata and Grafton [30] state that "when music engages the human mind most strongly—when performers play music, or when listeners tap, dance, or sing along with music—the sensory experience of musical patterns is intimately coupled with action." Whether listening to music or playing it, the body engages just as the mind does. Concepts such as muscle memory and embodied cognition, discussed below, are well understood with respect to music.

Theories of embodied cognition are the reaction to the notion of mind-body dichotomy, which has been the dominant view of brain-body interactions for many years. In this view, the mind is a completely distinct entity which performs cognition independently of the body: the brain is the site of creativity and reason, while the body acts as a machine that holds essential organs. Increasingly, scientific evidence is questioning this separation. Theories of embodied cognition suggest that cognition occurs not only in the brain, but also in the body. Cognition consists of one large, inter-connected system of cells which connects the brain and all of the sensory organs [35].

This view is consistent with the concept of muscle memory, a phenomenon well understood in both academic research and individual experience. Muscle memory refers to the ability of an individual to recreate a learned series of movements without conscious effort. Different kinds of memory have been suggested, with procedural memory referring to the ability to remember and carry out physical activities, and perceptual memory referring to the recall of sensory input linked to specific motor

skills. These kinds of memories work together to allow an unconscious recall of physical movements. When an instrumentalist interacts with their instrument, for example, they are forging procedural and perceptual memory, both of which contribute to the formation of muscle memory and embodied knowledge. [43]

2.3.2 Cognitive Experience of Live Coding

Live coding is an unorthodox way to physically engage with music. In contrast to traditional music performers, live coders navigate abstract ideas such as algorithms and concepts, rather than navigating a tightly coupled connection between sound and gesture.

These two approaches may seem completely disjoint, but there is physicality inherent to both practices. Malloch et al. [37] consider a spectrum of performance modes along an axis of real-time involvement, which they label as "interruption tolerance." Live coding and many gestural performance styles, such as acoustic instrument performance, are on opposite ends of this axis. Live coding operates on a symbol level, with a high degree of interruption tolerance, while acoustic instrument performance operates on a signal level, with a low degree of interruption tolerance. They are not wholly disjoint approaches. They do, however, employ embodied cognition in different ways.

Baalman discusses how the live coder and the computer form a system that interacts with each other to translate concepts into code [3]. Figure 2-4, a recreation of a figure from Baalman's paper, identifies not only the brain and the body to be one cognitive system, but the computer as an extension to this system. These systems interface through concepts, the human motor system, the physical computer interface, then the computer software. As a live coder engages with the computer, the computer engages back. Baalman states "... the longer we spend programming in a particular language, the more our mind gets attuned to the language and shapes our thinking about concepts to express. Not only our mind gets attuned, also our body - as we type certain class names, our fingers can become attuned to particular sequences of letters that form words that have meaning in the code. As such code is

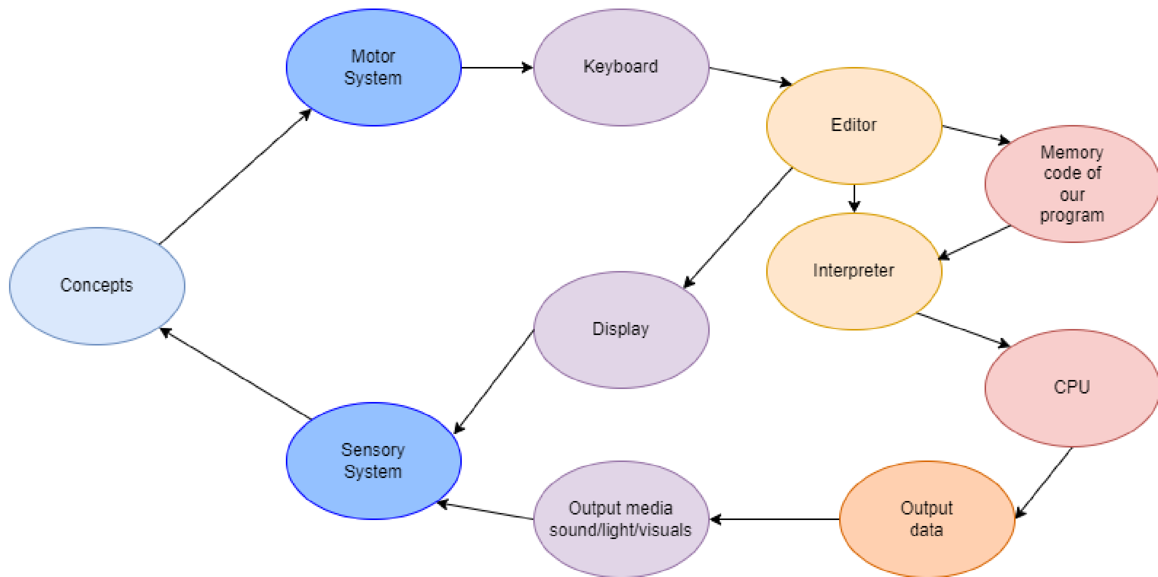


Figure 2-4: Human/Computer interaction feedback loop, a recreation of the figure from Baalman [3]

an embodiment of our concepts." The design of the system is not a trivial component of the live coder's engagement with it: it informs the way that the live coder embodies knowledge about it, and subsequently, how that knowledge is used.

However, there are distinct differences in the extent to which a live coder embodies knowledge and to which an acoustic instrumentalist does. Sayer [47] suggests: "The overhead of encoding motor skills for live coding is less burdensome and consequently the cognitive resource gains of 'automaticity' are reduced compared with instrumentalists. It is true that the act of typing is a motor skill but there is less opportunity to build complex amalgamations of primitive units that facilitate direct control over the sound elements in a rich parameter space. That's not to say that fine control over sonic material is not available to live coders but the means of accessing it is more heavily reliant on process memory than pre-encoded units of behaviour from object memory. ... I am suggesting that live coders may be less susceptible to the influence of habit and mechanical modes of musical expression because the motor skills required are less complex than instrumentalists and consequently less of their behaviour is activated outside of their perceived conscious control or awareness." Because live coders do not have large amounts of physical pre-encoded behavior to draw

on, they are not able to completely relinquish conscious control, despite their ability to become proficient at typing and familiar with the code structures available. This requirement of conscious control may hinder a live coder's ability to enter a flow state [2], which has been recognized as a significant determining factor in achieving flow [52]. However, it is unclear to what extent a computer interface may inhibit flow, if at all.

2.3.3 Previous Gestural Live Coding

Although the field of gestural live coding has not seen much attention, a few projects have considered this cross-section.

Wezen

Marije Baalman's series Wezen [5] explores the relationship between gesture and music. In one particular piece of this series, Gewording, Baalman transitions between playing the instrument via gestural controllers on her hands, and live coding the way that these controllers are mapped. Baalman discusses this interaction: "My role shifted throughout the whole process, from being the actor and mover to being programmer.... This means that, since I embody movements and listen to many variations of their sonification, I am aware of the kind of data that particular movements will create. I can incorporate this knowledge into programming a sound and its mapping to the sensors. At the same time, I create new scenarios by code in which I can explore my movements and their effect on the sound. The sound informs my movement, which in turn informs me in my subsequent mapping decisions." [4]

Baalman's interaction mode, transitioning between using gesture and live coding during a performance, is an example of successfully playing live code with gesture.

Live Coded Instrument

In their paper "Live Coding the Mobile Instrument" [32], Lee and Essl consider how gestural interaction can be effectively introduced into live coding, considering the

[2] Flow is an established psychological phenomenon first posited by Mihály Csíkszentmihályi [15]

difficulty of achieving "immediacy with live coding as if one would play a traditional musical instrument." Their approach involves decoupling the live coder from the instrumentalist, so that the live coder is real-time programming the instrument while the instrumentalist plays it, with a goal of incorporating the "instrumental virtuosity and expressivity" into live coding. They successfully implemented and performed this system, suggesting promising results for further gestural engagement with code.

The next chapter will explore more implications of bringing gestural control into a live coding practice, drawing on the research done in this background, and considering new conclusions for how a live coding practice may be able to incorporate gestural control.

Chapter 3

Conceptual Analysis

The purpose of this chapter is to investigate existing frameworks that relate gestural musicianship to live coding. The resulting analysis guides the decisions and reflections made throughout this thesis.

3.1 Features of Music Performance

This section considers several aspects of music performance in both live coding and gestural music practices. For this thesis, a gestural musician is defined as a musician who engages with a physical interface that directly translates gesture into sonic output. The most common example of a gestural musician is an acoustic instrumentalist, which informs most of the analysis done in this section.

3.1.1 Embodied Knowledge

Embodied knowledge is a nearly ubiquitous element of music performance. A performer may have concrete intellectual ideas about a performance, but they must actuate a physical interface to realize these ideas. For example, a pianist may want to play an A chord and must press keys to musically express this idea. Depending on the musical interface, a performance's reliance on embodied knowledge varies. In most acoustic instrumental performances, embodied knowledge is crucial. An acoustic

instrumentalist practices thousands of hours in order to develop the muscle memory required to express complex ideas through the instrument. It is only through this fluid physical recall of practiced structures that a trumpet player knows how their fingers and breath must coordinate to create a desired note.

However, embodied knowledge is more than the muscle memory required to play a note. Musical ideas may originate from the embodied knowledge rather than a conscious cognitive process. For example, it is unlikely that a bebop musician carefully chooses every note in a complex improvised solo. There are physical limits on conscious monitoring: an analysis of jazz solos has shown that the limit to improvisational novelty is about 10 actions per second [47]. Although a musician may consciously consider higher level ideas about the structure of the solo, the individual notes may be chosen by lower-level physical processes. This organization can be very powerful, because the performer is able to generate expressive and complex outputs through making physical choices on several levels, both conscious and unconscious. In this way, the performer is a complex and carefully trained system, with knowledge inputs from many different sources.

Live coding as a musical practice also relies on embodied knowledge, though less obviously. The performer interfaces with a musical system through a computer keyboard, and through the language of code. When a live coder considers an input to the system, they must translate it into code. This process is accomplished through the coder's familiarity with the keyboard as well as the structures available in their programming language. For many programmers, writing a for loop is done through muscle memory, and the ease of typing words on a keyboard enables quick transduction from an idea into a real input to the system. Unlike an acoustic instrumentalist, live coders have the benefit of thousands of hours of practice on a computer keyboard just from day-to-day life. However, this embodied knowledge plays a smaller part in the performance. The live coder does not experience the same immediate sonic response from embodied knowledge as the trumpet player described above, who can choose notes unconsciously. Inputs to the system are almost always consciously chosen and carefully encoded into the codebox, even though this process is facilitated by

muscle memory.

The impacts of these differences are vast. Different physical interfaces demand different types of embodied knowledge, and in addition to influencing the way that the performer's ideas are carried out, they influence the way that the performer thinks about the system. A pianist and a trumpet player have different conceptions of what they will play, as the way that they can carry out sonic production differs. The differences between a pianist and a live coder are significantly larger.

3.1.2 Coupling to Time/Real-Time feedback

In gestural performances, the performer's actions are directly coupled to real-time sonic output. A gestural musician recognizes that any adjustment to the system will result in an immediate change in sound. Additionally, some gestural musicians, such as acoustic musicians, are fully responsible for generating sounds: if the musician stops playing, the music stops. This incurs a constant cognitive load.

Live coders, by contrast, may perform actions that result in an immediate sonic response, but the majority of their actions will not. Typing into the codebox itself does not change the sound, and even running a line of code may not result in aural change until the code takes effect. For example, changing a note in a sequence will not produce a sonic outcome until the associated step of the sequencer plays. The asynchronous system changes are possible because the system plays automatically once the live coder runs the code. This frees up cognitive capacity, as the live coder is not responsible for generating real-time sounds.

Increased cognitive capacity enables more complex thoughts and awareness of the system, but it comes at a cost of decreased involvement in each sonic output. A gestural musician may not have the capacity to consider as many high-level architectural choices for the music, but they will have tightly-coupled and intuitive control over every sound; they possess more embodied knowledge of the system.

3.1.3 Precision and Accuracy

Live coding can be considered precise because live coders have the ability to absolutely and specifically control the algorithms creating the music. It is difficult for a gestural musician to recreate a sound exactly, but it is trivial for the live coder, who can specify the exact state and easily reproduce it.

However, gestural musicians may perform with more accuracy. Often, gestural musicians can rely on their system being fixed, meaning that an input to the system will predictably elicit some sonic output. A pianist is never surprised that pressing a C4 key will produce a C4 with a volume corresponding to the force of the key press. Because a live coder creates the environment that they are interacting with, they often do not know how an input to the system will affect the sound. Despite the exactness of code, the intention of a new input into a live coding system may not be exact at all. The consistency of a gestural musician's system may enable stronger embodied knowledge, as the strong input/output patterns are observed repeatedly.

Inaccuracy is not necessarily a hindrance to the performer. Unexpected sonic outcomes can alert a performer to the possibilities of a performance. Just as acoustic musicians have found new ways of playing instruments through accidents and mistakes (such as the discovery of the altissimo range of a saxophone), live coders can discover new features of their system by experiencing unexpected outcomes. These occurrences can also create a strong sense of liveness: the audience is witnessing the novelty of the system alongside the live coder.

3.1.4 Exploration

The inexact intentions of a live coder's input can allow for a different mode of playing, with a focus on exploration rather than exact intentional generation. A live coder may try different inputs into the system to experiment with different sounds, because it is difficult to have an exact mental model of what each change will do. This mode of playing allows for a different kind of performance.

Gestural performance also allows for sonic exploration. Despite the predictability

of a gestural instrument, other factors may be unpredictable, such as other musicians' choices in a jazz combo. A pianist may decide to test a different chord voicing which could change the way a soloist's notes sound. Additionally, a gestural musician can try new gestures during a performance. However, the mental bandwidth consumed by gestural performance may detract from the musician's ability to plan experimentation, and because of the likely static nature of a gestural musician's system, the outcomes of exploration may still be more predictable.

3.1.5 Composition

In part due to the cognitive load relief of performing with a live coding system, a live coder can take on a compositional role. Because they do not need to focus on every sonic output, they can create and adjust large structural elements of the sound. A live coder can create and structure a sonic environment with an unlimited number of different voices, each chosen by the live coder.

Gestural musicians can also compose, but to a lesser extent. There are fundamental limits to the number of voices they can play and the amount of cognitive bandwidth they can supply to each one.

3.2 Combining Approaches

Gestural musicians and live coders operate in very different modes of performance. While initially these differences may seem irreconcilable, they are not completely distinct from each other: for each of the criteria explored above, they exist somewhere on a spectrum. A large goal of this thesis is to explore the space between these approaches. Malloch [37] presents a spectrum of music interaction along the axis of real-time involvement, labeled "interruption tolerance," to indicate how immediately an interruption will propagate into the sonic output of the system.

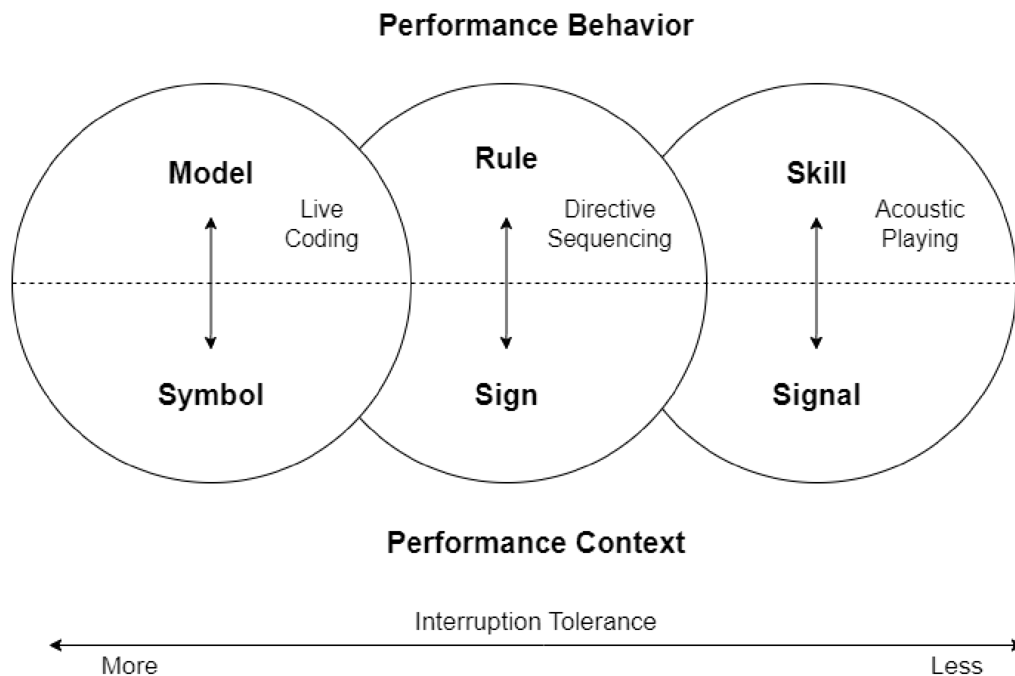


Figure 3-1: The Symbol/Sign/Signal axis from Malloch [37]. The significant examples are included in this recreation of Malloch's figure.

Live coders handle symbolic representations of musical concepts, while gestural musicians often directly manipulate the signals that correspond to the sonic output. Though they are at different ends of this axis, the continuum suggests that these approaches may not be unbridgeable. Interestingly, sequencing appears between these two endpoints, suggesting potential for bridging the two.

3.2.1 Goals

Many aspects of gestural performance are powerful musical tools: the performer's ability to engage physically with the sound of the performance allows for easy expressivity of emotion. Strong coupling of physical interaction with immediate sonic output gives gestural musicians a way to forge strong muscle memory pathways that can deepen the performance, for both the performer and the audience. Many electronic music performers have achieved similar interactions as gestural musicians through the use of MIDI controllers. Through MIDI controllers, it may be possible to make the unique

aspects of gestural instrument performance available to live coders:

1. Increased embodied knowledge of system: through use of a MIDI controller, a live coder could use gestural interaction during a performance. This could enable the live coder to develop a physical sense of the system, recruiting their body to encode subconscious knowledge. This knowledge may add to the depth and expressivity of the performance, as well as facilitate more intuitive interactions.

2. Real-time feedback: physical controllers are designed for real-time feedback. It is very intuitive to turn a controller's knob, for example, and gauge how the amount it is turned affects the sound through immediate aural response. Using a physical controller may allow live coders to sense algorithms, similar to how acoustic instrumentalists sense the complex processes of their instrument. In both cases, it may not be necessary for the performer to exactly understand the structure of the system for them to gain an intuitive sense of how perturbing the system will affect its sound.

3. Physical exploration: live coding is unique in the extent to which it allows performers to explore the system. By alleviating the real-time demand of generating in-time sounds, the performer is free to use high-level conscious processing to consider ways to navigate the system they have created. Beyond exploring the sonic space through written algorithms, a live coder may be able to use a controller to use physical motions to interact with the sonic space. Physical interactions are satisfying for both the performer and the audience; people are used to exploring the world physically. This could change the way that the coder conceives of changing, or creating, the system. This may allow for entirely new modes of performance.

4. Parallelism: a live coder can only change one aspect of the code at a time, by manipulating the code at the cursor position. Gestural musicians can engage with several different aspects of their system simultaneously, e.g. a wind player

can change the note, timbre, volume, vibrato, etc. simultaneously. A physical controller may allow a live coder to engage in code similarly, adjusting multiple code elements at the same time through different simultaneous controller actuations.

3.2.2 Challenges

Although the intersection between these domains is promising, there are several inherent challenges.

1. Conceptual Model: Live coders and gestural musicians have different cognitive models of a performance and different methods of performing. The thought processes of a gestural musician, who must focus on the moment-to-moment sonic production of the system, are starkly different from those of a live coder, who is freed from that mental load and has a very different kind of control over the system. Trying to find a middle ground between these disciplines may lose the benefits of both.

2. Different Performances: Electronic music performances often emphasize different features than acoustic instrumental performances do. For example, carefully timing musical events to create sonorous sounds may be of little interest in a live coding performance, in which the structure and timing of events are less rigidly expected by both the performer and audience. The introduction of real-time physical interactions, which are crucial and beneficial in acoustic instrumental performances, may not add to a live coded performance.

3. Changing System: Many gestural instruments always stays the same. Every performance, an instrumentalist knows how their instrument will respond to different inputs. However, in a live coded system, the way that a controller is mapped may change across different performances or even throughout the same performance. This complexity may make the development of intuitive physical control and muscle memory very difficult, and may consequently confuse

the performer and add additional cognitive load. It may not be possible to gain embodied knowledge of what different controller interactions will produce through the duration of a performance, making the controller useless.

4. Intended use of interfaces: a computer keyboard is optimized for writing characters into an interface, making it ideal for writing and manipulating algorithms. This is how a live coder can have such precise control over the algorithms in the system. Manipulating algorithms through a physical controller that was meant for tuning stable parameters and sending note messages may be unintuitive and clunky. Live coders may rather opt for the interface that live coding was designed for.

5. Practice of Live Coding: Live coding is often considered an experimental art form that pushes the boundaries of conventional music-making. One important component of a live coding performance is transparency. The TOPLAP live coding manifesto draft [49] states: "Obscurantism is dangerous. Show us your screens." Involving a physical controller may detract from this notion. Live coders may want nothing to do with a physical controller, as using one may be counter-intuitive to the purpose of live coding.

These challenges and potential benefits inform the implementation decisions made in GALiCA, as well as the resulting analyses. Chapter 5 utilizes this discussion for guiding the four implemented gestural approaches.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

The Seq Class

This chapter discusses the conceptualization and implementation of the GALiCA sequencer. The motivation of this section is to present a sequencer that optimizes for flexibility and encourages creative and intuitive gestural interaction with live coded algorithms.

4.1 Sequencer Overview

Most live coding systems are built around sequencing sonic events in time. This process is often conceptualized with sequencers. Sequencers play a significant role in the history of electronic music, and their ubiquity and familiarity to electronic musicians makes them an ideal structure in live coding. Many live coding systems, such as TidalCycles and Gibber, use a sequencer as the conceptual model for a class, which the live coder uses to generate sound. Because of the universality of this approach, and its wide range of expressive capabilities, a sequencer class was chosen for GALiCA as well.

Fundamentally, a sequencer is anything that can schedule events in time. Sequencers typically have a series of steps that are indexed into. When a counter that increases linearly with time (a clock) tells the sequencer that it is time to execute a step, the sequencer will call an event based on the value of the current step. The index is then incremented for the next step.

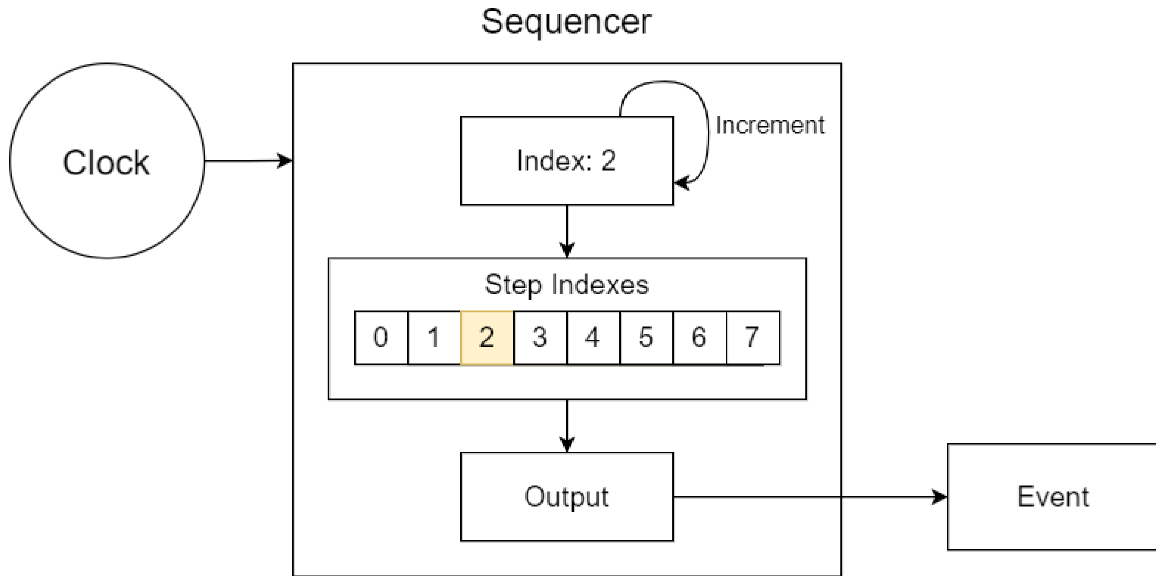


Figure 4-1: A basic sequencer executing a step

Figure 4-1 shows the process of executing a step. The clock sends input to the sequencer and triggers the step. Then, the sequencer selects a value based on its current index and uses it to determine the sequencer's output for that step. After the output is computed, the sequencer executes the corresponding event (such as sending a MIDI note). Finally, the index increments to prepare for the next step.

4.2 GALiCA Seq

One of the primary goals of the sequencer implemented in GALiCA's Seq class is to enable users to quickly and easily begin making music with the system. Users are likely already familiar with sequencers, so using this concept to inform interactions with the system may enable users to understand it quickly and intuitively.

The other primary goal is flexibility over what each Seq instance does. However, a maximally flexible sequencer would be a blank codebox: there necessarily must be a model based on a sequencer's intended use that users can adapt to their needs. Otherwise, users may not realize the creative potential of the Seq class. This will take the form of some intended constraints. However, every variable and function in a Seq instance can be overwritten. Some aspects are intended to be overwritten while

others are not, but the user has full control over how each instance operates. That is how the Seq class balances flexibility with useful abstraction: creating a model for how a Seq instance can be used, but allowing users to modify it however they want. This key idea is considered throughout the design choices of the Seq class.

Ultimately, the purpose of a GALiCA sequencer is to use this flexibility to enable intuitive interactions with a MIDI controller, discussed in detail in chapter 5.

4.2.1 Overview

The elements of a sequencer described above are described in this section.

Clock

The timing of every Seq instance is governed by a global clock, which monotonically increments throughout the lifetime of the program. There are two options for increasing the global clock over time:

1. Internal tracking: the clock increments internally based on timekeeping done through a JavaScript worker.

2. External tracking: the clock increments based on MIDI clock messages supplied by an external program.

Every time the global clock increments, each Seq instance checks if it is time to execute a new step.

Index

When a Seq instance receives a signal to execute a step, the next step value is selected by indexing into an array of values, using the current index. After the step has been selected, the index is incremented. By default, as with most sequencers, the index increments by one, so that the following step is called next. However, many sequencers allow for different index increments. In a GALiCA Seq instance, the user can define both the index increment and the function that performs the increment.

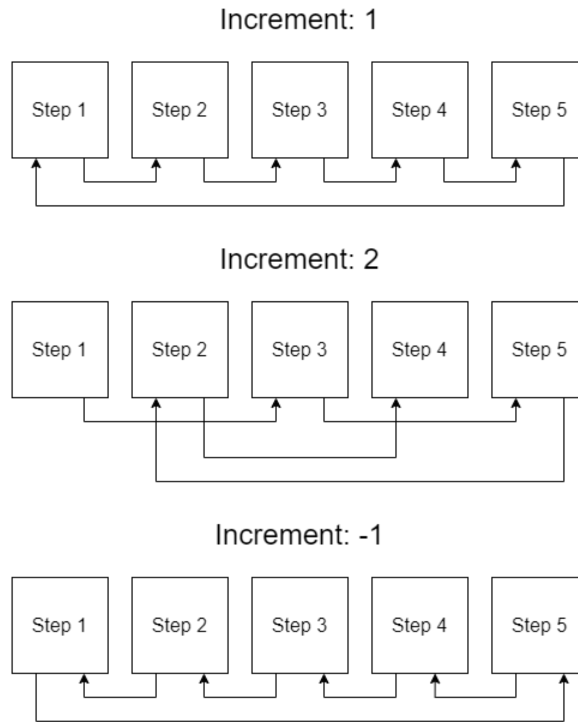


Figure 4-2: Examples of different increment values stepping through an array

With the default increment of 1, the values are stepped through sequentially. When the increment is changed to 2, the step immediately following the current step is skipped, and when the increment is a negative number, the array is stepped through backwards.

Allowing for different increments gives the user more control over how each Seq instance executes, and it invokes ideas from familiar hardware sequencers such as the Arturia BeatStep Pro [2].

Output

Although the most common output for a sequencer is a note event, a traditional hardware sequencer simply outputs a signal corresponding to a value. This opens up endless possibilities for what a sequencer can sequence. To leverage this inherent flexibility in a sequencer, GALiCA allows users to output any event of their choosing. By default, this is sending a MIDI note message, but it can be anything. This is explored in deeper detail in the section 4.3.2.

4.3 Implementation Details

4.3.1 Values and Durations

The simple model of a sequencer described above does not discuss the amount of time between each step. In some traditional sequencers, each step has the same length, but the ability to adjust the length of each step has become more prevalent with newer sequencers. GALiCA utilizes this idea to enable users to define the durations of each step. This includes setting each step to be the same desired length as well as using an array of durations that is indexed alongside the values array. This idea draws inspiration from the Gibber Seq class [23].

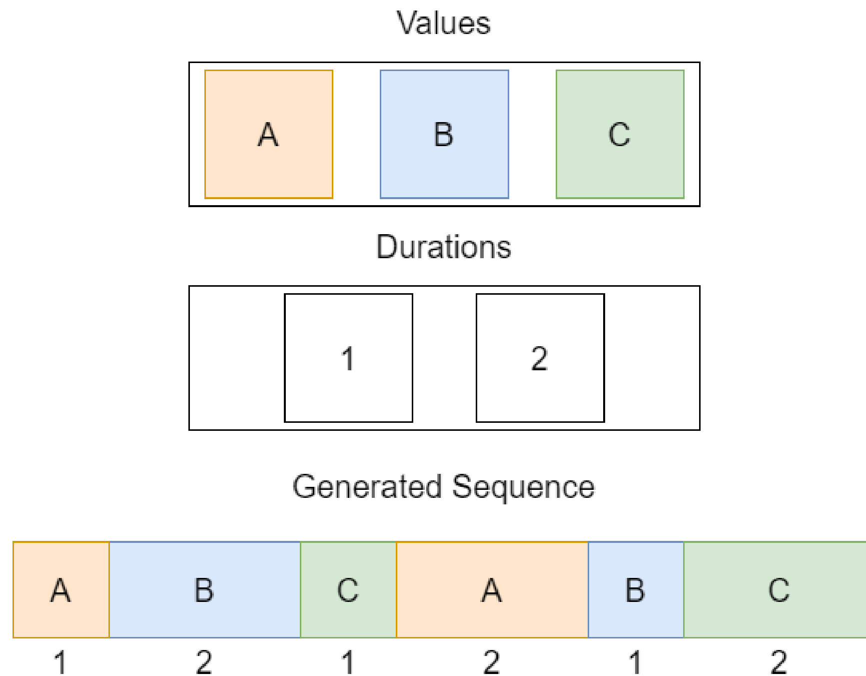


Figure 4-3: An example of a sequence generated by separate arrays for values and durations.

In figure 4-3, both arrays are stepped through with an increment of one. For the first step, the value A is called and has a duration of 1. Both of these indexes are incremented, resulting in a next step value of B and a next step duration of 2. Once one unit of time has elapsed since step A was called, the next step occurs, so B is called with a duration of 2. This pattern continues, resulting in the generated

sequence shown.

The separate instantiation of values and durations allows users to choose exact values for each, and the ability to separately control rhythm and value allows users to easily produce extended repeating phrases. However, this choice comes with the challenge of no rhythmic bar constraints, leaving it up to the user to count out durations and determine the total time of a Seq instances' durations loop. This increases the difficulty of keeping multiple sequencers in phase with each other. Because a primary design consideration for the Seq class is flexibility, the freedom of separate values and durations arrays was chosen despite this inconvenience[1].

4.3.2 Functions

A GALiCA Seq instance has a variety of functions that are called to carry out sequencing. Additional functions are added to the control flow to enable increased modifiability of each Seq instance. The functions called during a step are shown in figure 4-4.

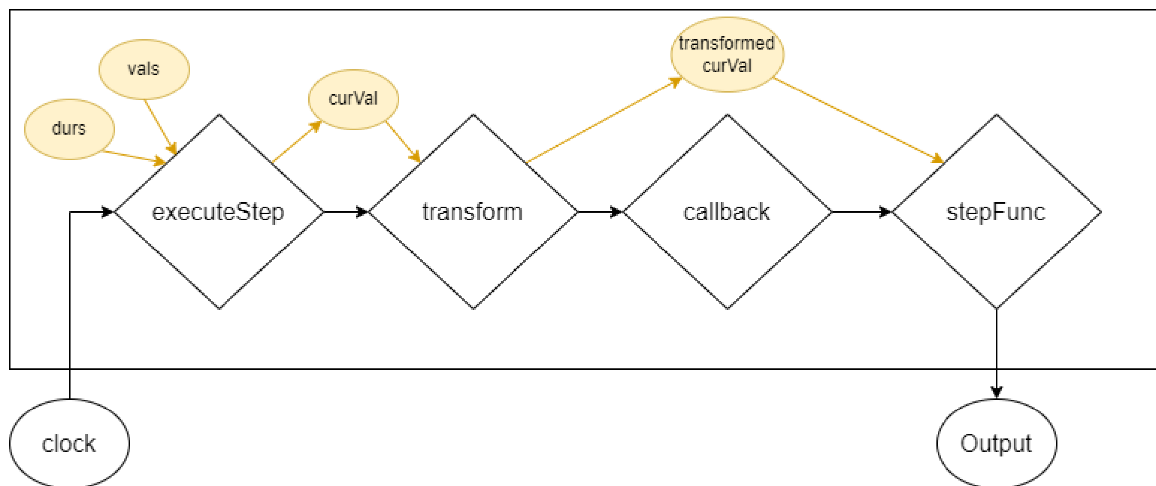


Figure 4-4: A flow diagram of the functions called during a GALiCA sequencer step. The variables are represented by yellow ovals, and the functions are represented by diamonds.

The values and durations arrays are shown as vals and durs. When the global clock increments, each seq objects checks if it is time to execute, referencing the previous

[1]See TidalCycles for a different approach to organizing sequencer values and durations [39].

step's duration. If it is, then the Seq instance calls executeStep, which calls all the functions necessary for a step. Two optional functions are then called: transform is called and can optionally change the value retrieved from the values array, and then callback is called which can perform any arbitrary function on each step. Finally, the transformed value from transform is passed into stepFunc, which creates the desired event for the step. These functions are described in greater detail below.

executeStep

This is the function called when it is time for a sequencer to perform a step. It calls the functions described below and updates the step's value, curVal. Although it is possible for a user to overwrite this function, they would have to rewrite the necessary code or otherwise call the necessary functions.

stepFunc

stepFunc is a variable that points to the primary function executed on each step. Typically, this function initiates an event which utilizes the step's current value (curVal). It can be set to any function; by default, it is set to sendNote(). The GALiCA Seq class includes two predefined functions which stepFunc can be set to:

1. sendNote() interprets each value in the vals array as a scale degree, based on a global defined scale variable, which corresponds to an array of MIDI note values. curVal is used to index into this array (negative values are allowed and will index backwards through the scale array), and the corresponding MIDI note is adjusted by the octave variable.

   Once the current MIDI note value is determined, sendNote() sends a MIDI note off message for for the prior MIDI note and then a MIDI note on message with the current MIDI note value.

   Two special characters modify this default behaviour. If curVal is set to an underscore (_), the step is identified as a rest. In this case, a MIDI note off value for the prior note is sent but no note on message is sent.

Similarly, if curVal is set to a pair of underscores (_ _) then the current step is identified as a tie. In this case no note off or note on message will be sent.

2. sendCC() behaves similarly to sendNote(): it uses curVal as the CC value (limited to the range 0-127) and sends a CC message to the corresponding channel.

Note that parameters required for these functions, such as velocity and channel number, are included in each GALiCA Seq instance as well. However, users can set stepFunc to any function and choose to ignore the functionality for MIDI or CC sequencing, enabling the Seq instance to sequence any parameter in the code. For example, one sequencer may sequence velocity values for another sequencer. This could be accomplished through the following:

```
mySeq.stepFunc = function(){
    otherSeq.velocity = this.curVal;
};
```

Now, every time mySeq executes a step, its curVal will set otherSeq's velocity.

transform

transform is a function that performs an operation on curVal. It receives curVal as a function parameter and returns a transformed curVal. By default, it returns the original value. It is called immediately after the new curVal for the step is computed.

For example, a user may want to raise a note by a fifth based on a condition:

```
mySeq.transform = function(x){
    return (condition ? x+7 : x);
};
```

If the condition evaluates to true, then curVal will be curVal+7 for this step. This function runs separately from callback and stepFunc, so all three functions can be edited without affecting each other.

callback

callback is a function that is called immediately before stepFunc. It does not return anything and does nothing by default. The purpose of the callback function is to allow for user-defined scripts to be run on every step, separately from the primary stepFunc. For example, if a user wanted stepFunc to be set to sendNote but also wanted to change the rhythm of the sequence based on some boolean condition, they may do the following:

```
mySeq.callback = function(){
    condition ? this.durs = 1/2 : this.durs = 1/4;
};
```

This does not affect mySeq's execution of a MIDI note on a step, which is done in stepFunc, while still allowing extra code to run with every step.

Note that there is little functional difference between these mutable functions, transform and callback, besides where they are called in the Seq class. A callback could just as easily change curVal by manipulating this.curVal. The purpose of including these separate manipulable functions is to encourage their corresponding uses. If a user wants to transform curVal, it will be most intuitive to do so through transform, and they may even want to modify the transformed curVal in callback. These design choices reflect the broader goal: flexibility within the Seq class through user-changeable design choices made with a conceptual intent. The transform function was made with one concept in mind, but the user can choose to engage with this to their desired extent.

4.4 Parsing

In order to enable certain sequencer operations, the code from the codebox must be parsed and interacted with. This is done through CodeMirror [25].

4.4.1 Recognizing Seq Instances

Adding Seqs

When a user executes a Seq instantiation, it would be inconvenient for the user to have to additionally start the seq or add it to a data structure of Seq instances. Through codebox parsing, the system will recognize a Seq creation and automatically add it to a dictionary of Seq instances. This dictionary is checked whenever the globalClock is increased. If any Seq instances in the dictionary indicate that it is time for a step, they will execute.

Persistent Variables

CodeMirror is locally scoped so that any variables declared within the codebox will not be available to the rest of the system. This is problematic, as the user may want to refer to variables they have previously run. To counter this, the system parses variable declarations and makes them globally scoped, allowing them to be referenced by the user once they are executed.

4.4.2 Redefined Input Arrays

A Seq instance can be created with a named array. For example, an array called notes:

```
notes = [1,2,3];
mySeq = new Seq(notes, 1/4);
```

If the notes list is redefined:

```
notes = [4,5,6];
```

This will not affect the vals of mySeq, because the notes variable now points to a different object, and mySeq is using the de-referenced object. This is a rule in JavaScript. However, a user may want to operate on an array that has been passed

into Seq instances and see those changes reflected in the sequencers. To accomplish this, the codebox is parsed to find where named arrays are used in Seq instantiations, and whenever one of those arrays is redefined, the new array object replaces the original one in each Seq instance in which it appears. Now, if the example above occurs, mySeq's vals will read: [4,5,6].

4.5 Reflection

The sequencer concept detailed in this section provides the means through which the gestural interactions in chapter 5 are explored. The ability to modify the operation of a sequencer, through changing not only the values and durations arrays, but also the changing the step increment, modifying the step function, and creating transform and callback code, opens up many opportunities to intuitively engage with the inner workings of the sequencer. The ease of interacting with these changes allows for a large space of modifications which require very little code from the user.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

MIDI Algorithm Interactions

This chapter details four ways in which a MIDI controller can be used to interact with algorithms written in the codebox. The motivation for this section is to develop and evaluate techniques that integrate gesture into live coding, to examine the potential of a combined modality.

5.1 Overview of Implementations

The four Algorithm Interactions described in this chapter are:

1. Controller-Created Ternary Expressions
2. Controller-Mapped Expression Variables
3. CC Variables and Callbacks
4. MIDI Note Callbacks

The following sections detail the motivation, implementation details, and analysis of each technique.

5.2 Ternary Expressions

Ternary expressions are a common technique that enable live coders to generate conditional algorithms quickly. For example, `(x > 0 ? x : 0)` will return x if x is positive, or 0 otherwise. This section explores using a MIDI controller to generate these expressions, instead of a computer keyboard.

5.2.1 Goal

As discussed in section 3, live coding encourages sonic exploration. However, there are limitations to this exploration. Algorithms must be precisely defined in a codebox. If the performer decides to explore and alter the system's sonic output, they must choose which specific elements to adjust and choose exactly how to adjust them. All of this occurs on a high cognitive level. The performer must conceptualize the algorithms and parameters, then choose which ones to modify, one by one. If a performer wants to change a value, they will have to select and enter a new chosen number, e.g. changing the statement `globalClock%5%2==0` into `globalClock%5%3==0`. Although this is a useful method of interacting with the code, it does not recruit many of the physical mechanisms that gestural musicians use to great effect.

This limitation could potentially be addressed by using a MIDI controller to write codebox input. The first iteration of this idea is enabling the performer to use a MIDI controller to create a ternary statement: a user could actuate CC dials and MIDI buttons to create a usable conditional algorithm in the code. This is a completely new type of control over the sonic production of the system. This makes it possible for algorithms to not have to be precisely defined and conceptualized on a high cognitive level. Instead, the performer can create an algorithm through a physically guided process, and may even be able to develop muscle memory in doing so. The sonic exploration of live code can become physical as well as conceptual. Because ternary statements are commonly used in many live coding performances, this technique may be a natural way to introduce tactile input into the code system.

5.2.2 Implementation

Conceptualization

Instantiating a ternary statement from a physical controller requires re-conceptualizing what a ternary statement is. Programmers are familiar with having the ability to create arbitrarily long ternary statements with a very large value/operator selection. Although most programmers likely type ternary statements from left to right, there is no requirement that one must do so. Instead, a programmer could very easily write the condition last, or modify the condition after having written the return values. There are many possible conceptualizations, but three possibilities were considered here:

Ternary Concept	**Features**			
	Backtracking	**Conditional Format**	**Conditional Expression Length**	**Return Expression length**
Rigid Left-to-Right	No	<L value> <operator> <R value>	3	1
Value/Operator Left-to-Right	No	<value> <operator> <value> <operator> <value> ...	≥1	≥1
Selector	Yes	-	-	-

Table 5.1: Main differences between ternary types
Note: "Length" is the sum of the number of values and the number of operators in an expression.

1. **The rigid left-to-right ternary**: In this conceptualization, a ternary takes the structure:

```
(<L value> <operator> <R value> ? <T value>:<F value>)
```

where "values" must be either a number or a variable. It is created from left to right with no backtracking. The limitations are evident: the conditional structure excludes single boolean variables as well as more complex expressions

(unless they are stored in a variable, such as a = x%3). Creating a ternary expression with this structure would involve a finite selection of values and operators which are cycled through and once selected, unable to be changed, so that the control will flow monotonically left to right.

2. **The value/operator left-to-right ternary**: In this conceptualization, a ternary takes the structure

```
(<expression 1> ? <expression 2> : <expression 3>)
```

   where each expression is modeled as a series of value+operator+value choices which can be arbitrarily long, or as short as just one value. There is still a finite selection of values and operators, but now there is an infinite space of possibilities for each expression, unlike the rigid ternary. This added flexibility comes at the cost of the assurance that the condition <expression 1> evaluates to a boolean value. This ternary model still requires a strict left-to-right control flow, with some ability to progress to selecting the next expression once the current expression is complete.

3. **The selector ternary**: This conceptualization may use a rigid or a value/operator structure, but the key difference is that the control flow does not need to be strictly left-to-right. A choice may be backtracked to and altered by selecting the location of a value or operator. The main creation may still be left to right, with backtracking only used in cases where a previously chosen value is changed, or the ternary may be initialized as (___ _ ___ ? ___ : ___) in a rigid format, and the user can select which box to begin populating.

The complexity of each of these approaches increases from 1 to 3. Flexibility, the ability to create many different ternary statements, is an important design consideration. The use of the physical controller must not become cumbersome with too many fine-grained choices: the most flexible system will be simply using the computer keyboard to quickly type a ternary structure. However, there is a lower threshold on

flexibility, past which the MIDI controller will not be a useful tool for creating ternary structures at all. If the MIDI controller constrains the possible space of ternary structures too much, the desired ternary statement may be impossible to produce, and the live coder may have to return to the computer keyboard.

Implementation

To create a ternary expression, the user begins by typing `startTern()` into the codebox. When they run this line, the codebox freezes input from the user, and replaces `startTern()` with the beginning of a ternary expression: `"( x."` The user can flip through a number of variables by using a CC dial from their MIDI controller. The dial range is split equally among all the available variables. For example, if there are 4 variables, then the dial values are mapped into quadrants, as shown in figure 5-1. Once the user has selected their desired variable, pressing any MIDI note will lock the choice and progress to the operator choices, which are mapped to dial values as described before, as shown in figure 5-2. The user continues to select values via the CC dial and continue on to progress through the ternary via MIDI notes until the expression is complete, at which point the codebox is unfrozen, and the user can begin typing into it again. This process is detailed in section 5.2.3. Using this general technique, 3 approaches were attempted:

1. Full if/else: The first iteration of this design took the form of a full, multi-line if/else conditional. When the user typed `startIf()` into the codebox, the `if` appeared at the bottom of the screen, and the values were selected via the CC dial/MIDI note method as described before. This was functional, but the use of a full if/else conditional is limited. Additionally, replacing the `startIf()` command with the expression is much more intuitive: once `startIf()` has been used to make a conditional, the `startIf()` command is no longer useful. In the worst case, it may clutter the screen or be accidentally started again by running a block of code. Also, choosing the location of the conditional is important, and the best way to do so is to type the "start" command in the desired location. It is much more intuitive to replace the command with an in-line expression

rather than a multi-line one, so ultimately, the ternary expression was chosen over the if/else.

2. Rigid left-to-right: The first ternary expression was implemented in the rigid left-to-right form, where each value can be a variable or a number, chosen from a small selection of options, and the operator is chosen from `{<=, >=, =}`. This rigid structure is only capable of producing a small set of ternary expressions. A user may want to execute a more complex structure, such as "x/6 > y%3", leading to the decision to implement a value/operator format.

3. value/operator left-to-right: The final implementation takes the form of a value/operator left-to-right ternary with alternating variable and operator choices and no limit on how large each subexpression can be. The user uses the CC dial to select variables and progress through the expression with MIDI notes, and each subexpression ends once the user chooses a space instead of an operator. For example, if the user has already entered `x + 3 < y`, then for the next operator, turned the CC dial until the space appeared and selected it, a "?" would be entered, and the next step would be a variable entry. Although this method allows users much more control over the statement structure, it can quickly become cumbersome. The more choices there are, the more carefully the user has to select. Additionally, the possibility for invalid statements increases.

4. Selector: There is no implementation for the selector ternary concept. Certain mediums may be conducive to a selector ternary structure, but a codebox makes implementing this concept in an intuitive way difficult.

5.2.3 Example

An example ternary for each of the implemented ternary conceptual structures, a rigid ternary and a value/operator ternary, are described here. In each case, a CC dial is used to select either a variable or an operator, where the range of the dial is split among the number of options. Figures 5-1 and 5-2, show four and three options

respectively, but a user could have many more.

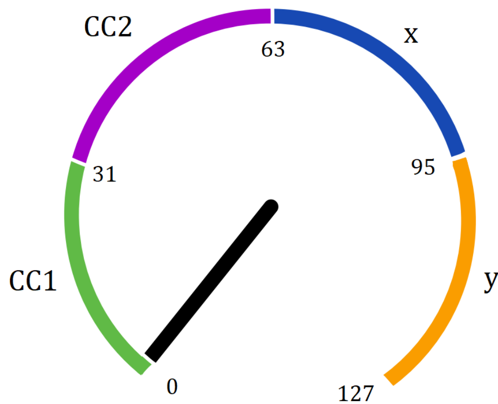


Figure 5-1: A dial selecting a value

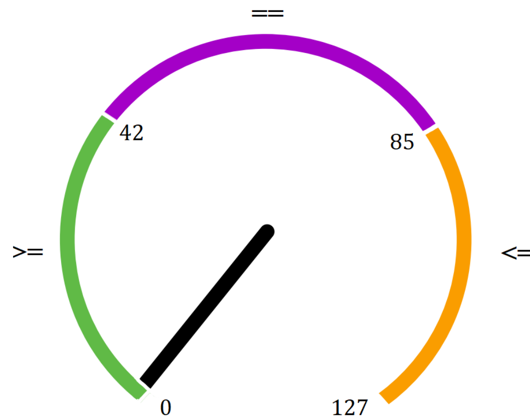


Figure 5-2: A dial selecting an operator

Rigid Ternary

In the rigid ternary conceptualization, a user would begin by typing `startTern()` and running the line.

```
startTern();
```

After running the line, the codebox freezes, and the `startTern()` statement becomes the beginning of a ternary expression:

```
(x
```

The user can then cycle through a number of variables and numbers using CC dials. Once the user selects the intended variable, CC1 in this case, the choice can be locked in by sending a MIDI message. Once the MIDI message is sent, the ternary expression moves on to allow the user to select a conditional operator:

```
(CC1 >=
```

Now, the same CC dial changes the operator type. Once the user lands on the

chosen operator, "==" in this case, and presses a MIDI note, they can now choose the variable to compare CC1 to:

```
(CC1 == CC1
```

Once they select the new variable, the ternary expression progresses by completing a "?" and allowing the selection of a new variable, which will be the returned value if the conditional evaluates to true:

```
(CC1 == x ? CC1
```

Now the user once again selects the value via the CC dial and continues through the expression by sending a MIDI note, at which point a ":" populates, and the user can choose the final value, which will be the returned value if the conditional evaluates to false:

```
(CC1 == x ? CC2 : CC1
```

Once the user chooses this value, the ternary expression is complete. The final parenthesis will populate, and the user will once again be able to type into the codebox:

```
(CC1 == x ? CC2 : x)
```

value/operator Ternary

In a value/operator ternary, a user would begin the same way (with a larger number of operator possibilities, such as + and -). The two approaches diverge once the second value is selected. Suppose the user has selected up to this point:

```
(CC1 == x
```

At this point in the rigid ternary, the expression would progress to the true return

value selection. In the value/operator ternary, however, the user can choose to either progress by selecting a space, or continuing the conditional expression with a new operator:

```
(CC1 == x +
```

Now, the user can select another value:

```
(CC1 == x + y
```

The user once again has the option to either complete the conditional expression by selecting a space, or continue the expression by selecting another operator. In this example, the user selects a space, and the conditional expression is complete. The ternary progresses to selecting return values:

```
(CC1 == x + y ? CC2
```

Now, the user can once again create an expression here by selecting an operator, or continue to the next return value by selecting a space.

```
(CC1 == x + y ? CC2 -
```

Once the user completes this expression, selecting a space will allow the user to select the final return value:

```
(CC1 == x + y ? CC2 - CC3 : x*(CC4+CC5)
```

Once the user selects a space, the ternary expression completes, and the user gains control of the codebox again.

```
(CC1 == x + y ? CC2 - CC3 : x*(CC4+CC5))
```

5.2.4 Reflection

The MIDI controller-created ternary expression was successfully implemented and provides a means to engage with codebox algorithms through a MIDI controller, which opens up new possibilities when performing with a live coding environment. However, there are several drawbacks to this approach that limit its usefulness.

First, the user may want to choose among many variable and operator options. For example, a user may want access to CC values 1-10, a few variables, and a variety of positive and negative integers. The CC dial is good at selecting among a few options; however, once the number of options exceeds this low threshold, choosing among them with a dial becomes difficult, as it is easy to overshoot the intended value. This can become very frustrating, and the precision of a computer keyboard may dissuade a user from using the MIDI controller to complete a task which can be done very quickly by traditional typing.

Second, in the current implementation, the user does not have the ability to backtrack, so mistakes are costly. If the user accidentally progresses after a mistake, there is no way to go back and correct it, so the ternary statement will have to be restarted. This could be fixed by including a means to go back and change values, i.e. implementing a selector ternary, but doing so would introduce more complexity, requiring perhaps different CC dials to be mapped to different actions (e.g. CC1 is "select variable" and CC2 is "select ternary position").

Third, externally editing the codebox is invasive and prone to errors. While creating a ternary structure, the codebox must be frozen, because changes to the ternary function code will break the ternary creation process: the current implementation uses regex matching to find the current algorithm and modify it. Giving the user the ability to invoke a command that freezes the codebox makes it necessary to give the user a way to regain control of the codebox. In the current implementation, this is done with a separate button. Having a button solely for this purpose adds unnecessary complexity to the user interface.

Fourth, unless one CC dial is dedicated to ternary expressions, CC dials will be

mapped differently throughout the duration of a performance. For example, the dial that sends CC1 messages may at certain points be used to alter CC1 values, and at other points be used to select ternary operators/values. This requires the user to shift mental models of the same controller which could become confusing during a performance. This is exacerbated by the mapping change being implicit: running `startTern()` changes the mapping in a way that is not immediately obvious. The alternative, in which one dial is set aside for ternary expressions, is limiting because, depending on the MIDI controller, this may reduce the number of available CC dials significantly.

Fifth, this technique does not have an intuitive performance flow. A user must type into the codebox to begin the ternary creation process, switch interfaces to use the MIDI controller to complete it, then likely return to the codebox to continue editing or run the expression. Without a strong incentive to creating a ternary expression via the MIDI controller, creating a ternary via the codebox seems more effective.

Sixth, there is no immediate sonic feedback from these MIDI controller interactions. A large contributing factor in forming physical associations through muscle memory is immediate feedback. Additionally, it can be more satisfying to the audience when a visible physical input to the system results in a noticeable change. Using a MIDI controller to silently add code to the codebox removes some of the potentially satisfying and important components of physically interacting with the system.

Given these drawbacks, the idea behind the MIDI controller-created ternary expressions is interesting, but it may not be practical for a performance. It addresses the ideas mentioned before: introducing a physical means of exploring the code and adding some amount of interest through human variability and error to the performance in a way that simply typing code cannot. Additionally, a user may be able to create interesting ternary structures very quickly through muscle memory. Associations formed through physical interactions can form embodied knowledge, and physically interacting with the system in another capacity may shift the performer's mental approach to using the system in an interesting way.

However, the learning curve to efficiently create structures through this method

is steep, and it is possible that even after spending time learning and practicing this technique, a user may still not approach the efficiency of simply typing out a ternary structure. Additionally, even if a performer becomes very skilled with this technique, they will still have to change interfaces, from computer (to run `startTern()`) to MIDI controller (to create the expression) and back (to run the expression). This process may be too awkward for the performer to utilize and too error prone to create practical ternary expressions.

5.3 Expression Variables

Rather than creating an entirely new algorithm through a MIDI controller— a task which may be most well suited to a computer keyboard— a MIDI controller could be used to select among existing algorithms. This section explores this possible approach to interacting with codebox algorithms through a MIDI controller.

5.3.1 Goal

Although creating ternary expressions through MIDI controller input has potential as a means of tactically engaging with code, there are several complications that can be improved on:

- Precision issues: It is difficult to create a useful ternary expression due to the precision constraints of MIDI controller input.
- Codebox editing: Modifying the user's code in the codebox is not ideal because it either requires suspending user input to the codebox or risking frustrating bugs.
- Unintuitive interface flow: The user must type on the computer keyboard, switch to the MIDI controller to create a ternary statement, then return to the keyboard to run the code and continue to edit it.

- Sonic response: There is no immediate aural feedback when manipulating the controller, which can be an important factor in both developing muscle memory and creating an engaging performance.

These limitations can be addressed by a different way of controlling algorithms in the code: using a MIDI controller CC dial to change which expression a variable in the code is mapped to. This would avoid the precise nature of actually implementing each algorithm (which is done in the codebox), remove the need to freeze user input to the codebox, and allow the user to dynamically adjust code to change the aural response without returning to the computer keyboard.

The motivation behind this implementation is to introduce more intuitive physical control over algorithms in the code. When a user is moving a CC dial and swapping out algorithms, they may begin to forge an association between a sound and the dial position, eliminating the need to mentally track the specifics of each algorithm. Rather than manipulating code through the conscious awareness of algorithmic details, the user may build embodied knowledge of the state of the system and use that to direct changes, basing these choices on an awareness of how a physical movement impacts the sound of the code. When multiple variables are mapped to CC dials, the user may create an even more intricate understanding of how dial movements interact with each other. This ability could enable a new space of rich interactions with the code that is not available from typing into the codebox.

5.3.2 Implementation

The user interface is now split into 3 screens: the codebox, a box showing what algorithm each variables is assigned to, and a box showing the available algorithms, shown in figure 5-3.

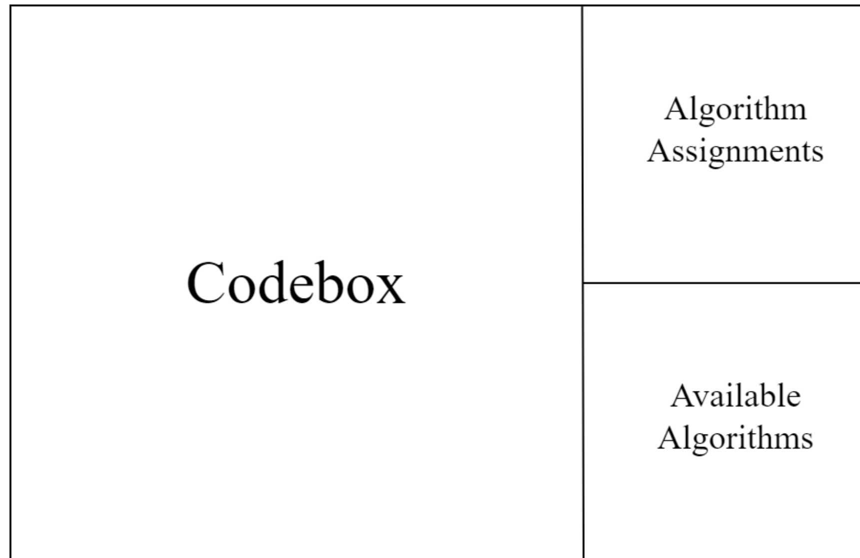


Figure 5-3: The layout of the GALiCA Expression Variable interface

The user can create a new algorithm by running the function `addToAlgs(expression)`, where `expression` is a string containing the desired algorithm. Although JavaScript allows variables to be set to expressions, the current implementation uses a string as input so that if the expression contains other variables, the result of the expression will update as they do. After the function has been executed, the new algorithm will appear in the Available Algorithm div.

To create a new expression variable, the user can run the line `assignAlg(varName, CCNum)`, where `varName` is a string containing the desired variable name, and `CCNum` is the CC number of the CC dial which will be used to control which expression the corresponding variable is mapped to. Once assigned, any adjustment to this CC dial will always change the variable's value.

Similar to the way that CC dials choose values as described in the Ternary Expressions section, the range of the CC dial is split into the number of available algorithms. If there is only one, then all values of the CC dial, 0-127, will map to that algorithm. If there are two algorithms, then CC values 0-63 will map to one algorithm, and values 64-127 will map to the other. As the user adds more algorithms, the corresponding range of values for each becomes smaller.

When a new expression is selected, the system will replace all instances of the

assigned variable with its corresponding expression: the code will run exactly as though the expression appeared wherever each variable does. The code specified in the codebox therefore does change based on MIDI controller input, but the appearance of the codebox does not change, removing the need to suspend user input to alter the codebox directly.

5.3.3 Example

A user might begin by choosing to change a conditional expression by a variable. First, they may add a few different conditions to the Available Algorithms:

```
addToAlgs("globalClock%6%5%2 == 0");
addToAlgs("Math.random() > 0.7");
addToAlgs("seqA.curVal == 5");
```

When the user runs this block of code, the Available Algorithms box will be updated correspondingly, as shown in figure 5-4.

Figure 5-4: The Available Algorithms box, populated with user-added algorithms

Now, if the user wants the variable `a` to contain a conditional expression, which

is mapped to CC1, they can run the line:

```
assignAlg("a", 1);
```

After the user runs this line, the Algorithm Assignments box will update, as in figure 5-5.

Figure 5-5: The Algorithm Assignments box, showing the current algorithm assigned to `a`

When the user turns CC dial 1, the variable `a` will map to a different expression, based on which expression corresponds to the range the CC dial is in. The mapping of CC dial 1 is shown in figure 5-6.

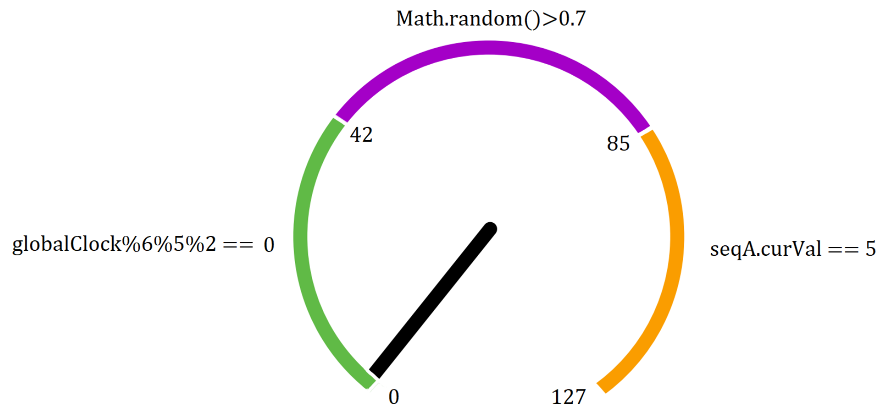


Figure 5-6: A CC dial selecting among the three available algorithms

The selected algorithm will update in the Algorithm Assignments box. For example, if the CC dial is turned to the top third of its range, the variable `a` will update to the corresponding algorithm, shown in figure 5-7.

Algorithm Assignments

```
a = Math.random()>0.7
```

Figure 5-7: The Algorithm Assignments box once `a` has been changed to a new algorithm

Now, the user can use the variable `a` in the code. For example, in the callback of a sequencer called SeqA:

```
seqA.callback = function(){
    a ? this.velocity=0 : this.velocity=120;
};
```

When the user moves the CC dial and changes which condition `a` is mapped to, that condition will populate the expression used in the callback. In this example, the condition will determine the rhythm of `seqA` by setting some notes to a zero velocity, so as the user moves CC1, the rhythm will change.

5.3.4 Reflection

This technique allows for a more intuitive way for users to adjust algorithms from MIDI controller input. It resolves one of the primary problems with algorithm creation that was uncovered by trying to create ternary expressions through MIDI controller input: the space of potential algorithms is infinite. An algorithm can take many different structures and use an large space of variables and numbers. This makes creating an algorithm through the limited controls of a MIDI controller very challenging. By using a MIDI controller to select among codebox algorithms, the creation of the algorithms is left to the codebox, the most reasonable domain for doing so. The user can create any of the infinite possible algorithms, but still interact with them through the controller.

This technique also provides immediate sonic feedback to changes in CC value. The user can intuitively interact with the MIDI controller and note how adjusting each dial changes the sound of the code. Through this, the user can continue an engaging performance solely by interacting with the MIDI controller, allowing for the physical exploration of the code without going back-and-forth between computer keyboard and MIDI controller.

However, there are a few inherent issues with this design. First, if a function calls one of these variables, and it is subsequently removed from the codebox, it will no longer change with CC input. The called variable must be on screen in order for it to change with the corresponding CC dial. This is due to the current implementation

and could potentially be addressed.

Another limitation with this design is that, currently, if the user wants to use expressions with different return types (e.g. a conditional expression which returns a boolean or an algebraic expression that returns an integer), some CC values will cause errors in the code if all expressions are listed under Available Algorithms. This could be addressed by having different classes of expressions, but it would add complexity to the system. Additionally, there is no compiler that checks whether an algorithm is valid at the time when it is added to Available Algorithms, as it is added as a string. This could cause frustration when the expression is put in the code and throws an error. A fix could be to compile the expression as a test and inform the user in the console if does not compile, but this is not currently supported.

Nonetheless, this approach has the potential to be an exciting new performance tool. A performer may use the controller to explore code as it is being written, or they may choose to do most of the live coding in the beginning of the performance, then switch to primarily playing the MIDI controller. Either performance mode would be a different way of engaging with a live coding system, and there is great potential to involve embodied cognition in these performances.

5.4 CC Callbacks and Variables

Traditionally, CC dials are used to set parameters of virtual instruments. However, there may be great creative potential in setting a variable in the code to a CC value, so that moving a CC dial changes that value in the code. This section consider the implications of this approach.

5.4.1 Goal

After a value has been written into the codebox, it only changes if either it has been set to a variable that changes over time (for example, `x` is sequenced by a sequencer), or it is deleted and replaced by the user who then reruns the line of code. In the first, the value change is totally automated by another process and no longer controlled by

the user. This can be positive as it frees the user's mental load, but correspondingly, it deprives the user from engaging with the parameter on a signal level.

In the second, the user has the freedom to write any value they wish very quickly. If, for example, they know that they want `x` to be 7 instead of 2, they can do so almost instantly. However, it is difficult to explore a range of values in this way. Every time the value is modified, the code must be manually rerun. If the user wants to try setting `x` to values between 1 and 20, for example, then the process of doing so is very arduous, and it requires the same repeated stereotyped movement: delete old value, write new value, enter line. This limits the way that variable modification can be physically embodied.

A new way to explore variable spaces is through setting the value of a variable in the code with a CC dial on a MIDI controller. Through this technique, the user can explore a range of values for a variable with physical movement. Additionally, each CC value can be mapped to a callback function, so when a CC value is changed, the code can rerun itself. Now, the user does not need to tediously enter and run lines of code: the changes can be done swiftly and physically. In this way, the user can form a physical understanding of how changes in a variable's value correspond with dial movement. The act of changing a value is physically embodied, rather than quantitatively noted. This may pull the live coding performance into signal level playing and give the user the ability to expressively interact with the sonic output real-time in an intuitive way.

5.4.2 Implementation

The implementation of this concept is simple: when the system receives CC input from CC channel x, it stores the received value in a variable called `CCx` and runs a callback called `CCx_func`. For example, if a message is received from CC channel 1 with the value 5, `CC1` is set to 5, and `CC1_func()` is called.

The user can write the variable `CC1` anywhere in the code, and when it is accessed, it will always equal the most recent CC data from channel 1. If `CC1` is used in an expression that is only called once, then the expression can be re-called in the

corresponding CC callback. For example, if a variable `x` is declared as `x = CC1 + 5`, and the user wants its value to update with changes to `CC1`, then the user can run the line: `CC1_func = function(){x = CC1 + 5;}`. Now, every time `CC1` is changed, `x` will change as well.

5.4.3 Example

To include a CC value in the code, a user simply needs to write its corresponding variable. There is no need to initialize it. Here is an example of a user setting a sequencer's transform function based on the current CC value from CC12:

```
seqA.transform = function(x){
    return x+CC12%5;
};
```

Rotating the CC12 dial on the MIDI controller will alter the value that is added to seqA's output every step.

If a user wants to set a sequencer's volume to a CC dial, they may write:

```
seqA.velocity = CC12;
```

Note that this will only update when the line is run. To make the line run every time a new CC12 value is received, the user can write:

```
CC12_func = function(){
    seqA.velocity = CC12;
};
```

Now, when `CC12` changes, `SeqA`'s velocity will change correspondingly.

5.4.4 Reflection

The implementation and use of MIDI-controlled CC variables is far simpler than the previously discussed MIDI controller-created ternary expressions and expression variables, but it may be even more powerful. Changing a value in the code is a very intuitive interaction, and the ability to engage with algorithms in a predictable but interesting way, while entirely away from the computer keyboard, opens up a new way to perform.

A user can map any value in the code to a CC dial: from note pitches and durations to more obscure values hidden in complex algorithms. A performer could physically participate in the code in a completely predictable way, such as mapping a CC dial to a sequencer's velocity, and perform more closely with how an acoustic instrumentalist may perform. Alternatively, they may choose more complex variables and participate in an exploratory way, such as scaling the received CC values to a sine wave or an exponential curve. A performance mapping could include some dials mapped to the former and others to the latter, enabling different modes of performance even just on the MIDI controller. As the performer builds an intuitive understanding of how the system sonically responds to movements of a dial, they could choose to use that throughout the performance and engage with it as embodied knowledge, or spontaneously change the mapping and force themself to start the process of learning the system all over again. This simple idea may have large implications for new ways to perform code. Assigning callbacks to CC messages is a traditional form of mapping CC dials within the NIME community, but because of the flexibility of live coding, there are many new possibilities in conceptualizing and implementing this technique.

5.5 MIDI Note Callbacks

CC Callbacks and variables are a creatively powerful tool. However, the use of a dial to initiate a callback suggests a continuous function, rather than a single event. One potential approach to allow a live coder to gesturally introduce a single event is a MIDI note callback.

5.5.1 Goal

When a live coder wants a new sound, they must run a new line of code. If the live coder wants to explore changes in sound, such as how a sequence sounds when its pitches are from a major scale versus when they are from a minor scale, they must manipulate and run lines of code. Although this process is a simple procedure for the live coder, its repetitive nature suggests the opportunity for gestural engagement.

The goal of MIDI note callbacks is to allow live coders to initiate events in the code, for example, switching between a major and a minor scale, by pressing a MIDI note on their controller. Similarly to CC callbacks, this may allow a live coder to form an embodied understanding of the algorithms they are executing, and work with them in a new way beyond the computer keyboard. This shift from symbol to signal level playing may have large implications for the performance style and cognitive approach.

5.5.2 Implementation

MIDI note callbacks are implemented similarly to CC callbacks. If the system receives a MIDI note on message with note value x, it runs a callback function called `midix_func`. For example, if the note on message contains a note value of 6, an F, then `midi6_func()` is called. Any octave of a note maps to the same callback function; this is to avoid complexities with MIDI controller octaves and ensure that the user can easily execute a desired callback. For example, if the note on message contains a MIDI note 66, corresponding to a higher octave of F, then the same callback is called: `midi6_func()`. These functions can be set by the user in the same way as described for CC callbacks.

5.5.3 Example

If a live coder wanted to change the scale of all sequencers in the project to be minor, they could do so by calling `setScale(minor)`. To switch it to major, they could run `setScale(major)`. The process of running these lines of code is not particularly

taxing, but a MIDI controller can do it instead.

To make a MIDI note C switch all sequencers to a minor scale, the user can run this line:

```
midi0_func = function(){setScale(minor)};
```

and to make a MIDI note D switch the scale to a major:

```
midi2_func = function(){setScale(major)};
```

Now, when the user presses any button on their MIDI controller that corresponds to a MIDI note C, the sequencers will play in minor, and if they instead press any MIDI note D, the scale which switch to major.

5.5.4 Reflection

The concept of a MIDI note callback is similar to that of a CC callback, but the way it is conceptualized is very different. Being able to assign a discrete event to a button push on a physical controller is a powerful tool for a live coder. In some ways, it challenges the mental distinction between a function and a signal, as the user can interact with functions as though they were signals. Rather than typing and executing code to initiate an event, the user simply pushes a button and the event occurs. New ways of conceptualizing functions may introduce new ways of interacting with the code.

With this technique, any event that can be expressed in code can be easily assigned to a MIDI note. This could be a low level event, such as changing the pitches in a sequencer, or a much larger scale event, such as using one button to transition to a completely different sound, perhaps by muting a large numbers of sequencers, switching virtual instruments for sequencers, and changing the tempo. This technique could be a powerful compositional tool, and the composition may be able to occur in a physically embodied way.

Chapter 6

Conclusion

The purpose of GALiCA is to examine the implications of combining live coding with gestural performance, to discover which resulting performance mechanisms are intuitive, useful, and worthy of further exploration, and to uncover less successful interaction modes which are less promising. The system was used as a testing grounds for four gestural approaches to algorithmic modification: ternary expression creation, expression variable selection, CC callbacks and variables, and MIDI note callbacks. Each approach incorporates gestural interaction through MIDI input from a standard MIDI controller.

This thesis also presents a sequencer class that was designed to facilitate the combination of live coding and gestural musicianship. The structure of GALiCA's Seq class includes functionality seen in other types of live coding sequencers, but each Seq object is also highly modifiable. Performers can edit a Seq object however they can imagine, and this is further facilitated by a unique conceptualization of sequencer interactions.

6.1 Sequencer Design Analysis

GALiCA's sequencer class was designed to allow for flexibility and modifiability while still providing conceptual clarity and a straightforward workflow. Constructing the Seq class to be highly modifiable is helpful for the gestural interaction potential of the

system: the ability to modify sequencers on-the-fly encourages new ideas for gestural interaction. All sequencers have a designed conceptual model. GALiCA's sequencer design was driven by considering which conceptual model will encourage the most creativity and showcase the sequencer's potential.

Because GALiCA is a live coded system, users have the capability to live code a sequencer from scratch and modify a Seq object however they want. However, complete open-endedness can be so free that the user may not realize the system's creative potential, and constructing desired structures from scratch may be time-consuming and cumbersome. After all, complete open-endedness would be a blank codebox. Therefore, a template sequencer class with a wide range of described interactions may encourage a high level of creativity and give the user the ability to interact with the system in a conceptually interesting way rather than an implementation-focused way. In other words, giving a user a useful conceptual model can be powerful.

Unique functions were introduced into the class, such as transform, callback, and stepFunc. These functions are already incorporated into each Seq instance's operation, so the user can redefine a function and not have to manage its place in the sequencer. These functions allow for the operation of a Seq object to be modified on-the-fly, showing the user a very large space of potential and easily implemented interactions.

In addition to these special functions, a number of default functions that enable quick creation of useful sequencers were also included. Important functions are shown in figure 6-1.

Many potential models were considered for the GALiCA's Seq design. There are numerous ways to conceptualize a sequencer, and this is just one of them. Overall, this model was successful for the purposes of this project, balancing typical use-cases of sequencers (such as sending notes by default) with open-ended modification potential that has a low overhead to implement.

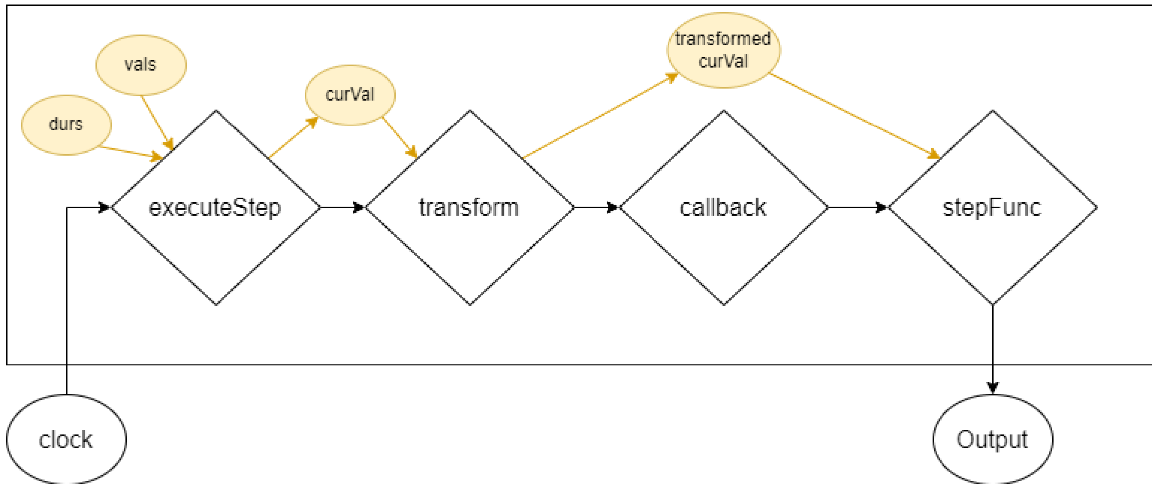


Figure 6-1: A flow diagram of the functions called during a GALiCA sequencer step. The variables are represented by yellow ovals and the functions are represented by diamonds

6.2 Gestural Interaction Analysis

Four different approaches were implemented to explore gesturally interacting with algorithms. These approaches had very different outcomes, with each suggesting a different performance style.

The first approach involved using a MIDI controller to construct ternary statements by cycling through available values and operators via CC messages. This idea came from the prevalence of concise ternary statements in live coding performances, which allow for quickly-created and interesting algorithmic patterns. Although successful in its execution, there were a few primary challenges with it:

- The performer would have to switch from computer keyboard to the MIDI controller to create the expression, then back to the computer keyboard to execute the code.
- The codebox had to be modified by the system in order to display the current ternary expression. This could increase the likelihood of program bugs.
- The gestural interaction did not elicit any sonic outcome that could be paired with movements, because code execution occurred after gestural input. This could hinder some of the cognitive benefits of gestural interaction.

Although a useful first attempt, this approach would likely not provide much performative benefit.

The next approach iterated on some of the shortcomings of the ternary statement construction. Instead of creating algorithms from scratch, this approach lets the user dynamically cycle through algorithms in the code that have already been written. This technique was much more successful, as the performer can engage with real-time sonic changes through physical manipulation and not have to switch between interfaces. Additionally, codebox editing is not required. Through this approach, a live coder may spend some of the performance solely playing code on the MIDI controller, although there are many potential performance techniques.

The final two approaches explore a completely different form of physical input: assigning code parameters to CC values, and code events to MIDI note events. Traditionally, live coders must redefine values in the code and re-run every necessary line if they wanted to explore how changing parameter values affects the code's sound. With this new approach, by assigning callbacks to CC and MIDI notes and using values tied to CC dials, performers can use gesture to real-time update values in the code and execute algorithmic events. This allows the performer to form associations between physical movement and sonic outcomes resulting from the algorithmic modification. The use of CC messages for continuous value inputs and MIDI note messages for discrete events could be used to great effect in a performance.

Analyzing these approaches reveals some characteristics of successful gestural engagement. If a performer wants to make a single change to the code, such as creating a ternary expression, they will likely prefer a computer keyboard. The benefit of gestural interaction in this case is not clear; a computer keyboard will be simpler and more intuitive to use. However, if a performer wants to continually change the code based on sonic response, they may find great benefit from being able to do so with a physical controller. A gestural controller allows the performer to scan through ranges of variables or algorithms, which is not accessible with a computer keyboard.

There are many potential styles of interaction using these gestural approaches. A performer may, for example, begin by live coding a system, then perform for the

remainder of the time on a MIDI controller. A performer may also switch back and forth throughout a performance to construct different sounds, or they may even use both nearly simultaneously, writing new algorithms and testing how parameters on the MIDI controller affects them.

6.3 Contributions

This project serves as an exploration into the possibilities and implications of gestural live coding through MIDI controllers, as well as a discussion of conceptual models for sequencers. The main contributions are:

- An overview of the cognitive and performative implications of combining gestural musicianship in the signal domain with live coding in the symbol domain.
- A new conceptual model of a sequencer that encourages modification of sequencer operation through novel re-definable functions.
- Functional examples of gestural control over live code algorithms and parameters, as well as a discussion of their implications on performance.
- Examples of using a MIDI controller to transduce physical interaction into useful algorithmic signals in a software system.
- A live coding system, GALiCA, that implements the approaches described above.[1]

6.4 Further Work

Beyond any concrete implementation goal, the overarching goal of GALiCA was to suggest the potential of gestural live coding. This approach has seen little attention in current live coding practices, and any success of GALiCA may show the performative potential of this new modality. Further work could more deeply consider the

[1]Source code at `https://github.com/hsavoldy/GALiCA` [46]

implications of this approach on performance and further evaluate its usefulness in a live coding practice.

Additionally, there is a large amount of work that can be done in exploring the conceptualization of a sequencer. A live coded sequencer can be conceptualized as anything, and the implications of this conceptualizations may be vast for a live coding performance. Further work may build off of the ideas presented here about what makes a sequencer concept useful, and create new interaction modes that open up more creative possibilities.

There are also implementation details in GALiCA that were not fully explored due to the scope of this project:

- Visual Feedback: using a MIDI controller as input to the code system could be benefited by some form of visual feedback. For example, displaying how values mapped to the controller are changing. This could be done in a way that avoids editing the codebox itself.

- MIDI Controller Profiles: providing a default controller mapping, or allowing users to save a mapping, could potentially enhance the experience of using a gestural live coding system.

- MIDI Flexibility: the choice to have the system send MIDI output rather than generate sounds itself allows users to choose any available external synth to generate sound, which can grant a lot of freedom. However, MIDI itself is a very limited protocol. Future work could be developed from this project in exploring uses of MIDI 2.0 [24].

Finally, the actual cognitive impacts of this new modality need further exploration. Research can be done as live coders become familiar with performing with a gestural live coding system, examining how their performances change, how their conceptualization of the system and their interaction style with it are affected, and if the benefits of gestural performance are seen in this new modality.

6.5 Conclusion

GALiCA provides a foray into an exciting potential direction of live coding, one in which a live coder is able not only to type into a codebox, but also participate in their code through gestural interaction, leveraging embodied knowledge of the system. Through a highly flexible sequencer class, GALiCA explores four different techniques for physically engaging with code. This work is a promising addition to gestural live coding, showcasing a few approaches that may completely change how a performer engages with a live coding system. GALiCA considers what is possible through a live coded sequencer and how physical embodiment can play a role in coding. Ultimately, it suggests the creative and expressive potential of a live coding system that incorporates gestural input to engage with live coded algorithms.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] S. Aaron. *Sonic Pi: The Live Coding Music Synth for Everyone*. https://sonic-pi.net. Accessed: 2023-11-22. 2023.

[2] Arturia. *BeatStep Pro*. https://www.arturia.com/products/hybrid-synths/beatstep-pro/overview. Accessed: 2023-12-05. 2023.

[3] M. Baalman. "Embodiment of code". In: *Proceedings of the first international conference on live coding*. 2015, pp. 35–40.

[4] M. Baalman. "Interplay Between Composition, Instrument Design and Performance". In: *Musical Instruments in the 21st Century: Identities, Configurations, Practices*. Ed. by T. Bovermann et al. Singapore: Springer Singapore, 2017, pp. 225–241.

[5] M. Baalman. *Wezen*. https://www.marijebaalman.eu/categories/Wezen.html. Accessed: 12/05/2023.

[6] J. Bischoff and T. Perkis. *ARTIFICIAL HORIZON*. CD. San Francisco, California, 1989.

[7] J. Bischoff et al. *The League of Automatic Music Composers 1978-1983*. Audio recording. New World Records. Cat. No. 80671. 2007.

[8] A. F. Blackwell et al. "Introduction to Live Coding: A User's Manual". In: *Live coding: a user's manual*. MIT Press, 2022, pp. 1–12.

[9] Buchla. *250e Dual Arbitrary Function Generator*. https://buchla.com/product/250e/. Accessed: 2023-12-05.

[10] Buchla. *The History of Buchla*. https://buchla.com/history/. Accessed: 2023-12-05.

[11] J. Chadabe. *Electric sound: the past and promise of electronic music*. Upper Saddle River, N.J: Prentice Hall, 1997. ISBN: 9780133032314.

[12] A. Clark. *Buchla 250e Dual Arbitrary Function Generator Tutorial & Demo*. Accessed: 2023-12-05. Apr. 27, 2016.

[13] N. Collins and J. d'Escriván. *The Cambridge companion to electronic music*. 1st ed. Cambridge University Press, 2017.

[14] N. Collins et al. "Live coding in laptop performance". In: *Organised Sound* 8.3 (Dec. 2003), pp. 321–330. DOI: 10.1017/S135577180300030X.

[15] M. Csikszentmihalyi. *FLOW: The Psychology of Optimal Experience*. Harper and Row, 1990. ISBN: 978-0-06-016253-5.

[16] R. B. Dannenberg. "Languages for Computer Music". In: *Frontiers Digit. Humanit.* 5 (2018), p. 26.

[17] Dazed. *What on earth is livecoding?* en. https:www.dazeddigital.com/artsandculture/article/16150/1/what-on-earth-is-livecoding. Accessed: 2023-12-05. May 2013.

[18] Doepfer. *System A - 100 Analog / Trigger Sequencer A-155*. English. Accessed: 2023-12-05. Doepfer.

[19] Doepfer Musikelektronik. *Doepfer A-155 Analog/Trigger Sequencer*. https://www2.doepfer.eu/index.php/en/item/a155. Accessed: 2023-12-05.

[20] M. Duignan, J. Noble, R. Biddle, et al. "A taxonomy of sequencer user-interfaces". In: *Proceedings of the International Computer Music Conference*. 2005.

[21] Electro-Music/Scott Stites. *KLEE Eurorack Sequencer Module*. https://reverb.com/item/34500774-electro-music-scott-stites-klee-eurorack-sequencer-module. Accessed: 2023-12-05. 2023.

[22] fonitronik. *Getting to Know Your Klee Sequencer*. Accessed: 2023-12-05. URL: http://www.modular.fonik.de/klee2/know_the_klee_draft3_203.pdf.

[23] *Gibber: Live Coding Environment*. http://gibber.cc. Accessed: 2023-11-22. 2023.

[24] J. Hamelberg. "Applications and Implications of MIDI 2.0". MEng. Thesis. Massachusetts Institute of Technology, June 2023.

[25] M. Haverbeke. *CodeMirror*. https://codemirror.net. Accessed: 2023-12-05. May 2007.

[26] Intellijel. *Metropolis Complex Multi-Stage Pitch and Gate Sequencer*. English. Version 1.30. Accessed: 2023-12-05. intellijel. May 12, 2020.

[27] Intellijel Designs Inc. *Metropolix - Powerful Multitrack Eurorack Musical Sequencer*. https://intellijel.com/shop/eurorack/metropolix/. Accessed: 2023-12-05. 2023.

[28] *International Conference on Live Coding (ICLC)*. https://iclc.toplap.org/. Accessed: 2023-11-22. University of Leeds, UK.

[29] H. Jackson. *MIDI CC List - Everything You Need To Know (Quick Guide)*. https://www.whippedcreamsounds.com/midi-cc-list/. Accessed: 2023-12-05. 2022.

[30] P. Janata and S. T. Grafton. "Swinging in the brain: shared neural substrates for behaviors related to sequencing and music". In: *Nature Neuroscience* 6.7 (July 2003), pp. 682–687. ISSN: 1546-1726. DOI: 10.1038/nn1081.

[31] J. Jenkins. *The History of Sequencers*. https://www.sweetwater.com/insync/the-history-of-sequencers/. Accessed: 2023-12-05. 2021.

[32] S. W. Lee and G. Essl. "Live Coding The Mobile Music Instrument." In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. 2013, pp. 493–498. DOI: 10.5281/ZENODO.1178592.

[33] LeoKliesen00. *File:Novation Launchpad S.png*. https://commons.wikimedia.org/w/index.php?curid=48087446. Accessed: 2023-12-05. Licensed under CC BY-SA 4.0. 2023.

[34] G. Loy. "Musicians Make a Standard: The MIDI Phenomenon". In: *Computer Music Journal* 9.4 (1985), pp. 8–26. ISSN: 0148-9267. DOI: 10.2307/3679619. (Visited on 11/02/2023).

[35] M. Macedonia. "Embodied Learning: Why at School the Mind Needs the Body". In: *Frontiers in Psychology* 10 (2019). ISSN: 1664-1078.

[36] Make Noise Co. *René - World's Only 3D Cartesian Music Sequencer*. https://makenoisemusic.com/modules/rene. Accessed: 2023-12-05. 2023.

[37] J. W. Malloch et al. "Towards a new conceptual framework for digital musical instruments". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. 2006, pp. 49–52.

[38] J. McCartney. "Rethinking the Computer Music Language: SuperCollider". In: *Computer Music Journal* 26.4 (Dec. 2002), pp. 61–68. ISSN: 0148-9267. DOI: 10.1162/014892602320991383.

[39] A. McLean. *TidalCycles: Pattern Language for Live Coding*. https://tidalcycles.org. Accessed: 2023-11-22. 2023.

[40] *NIME Proceedings Archive*. https://nime.org/archives/. Accessed: 2023-12-05.

[41] M. Noise. *René User Manual*. https://makenoisemusic.com/content/manuals/renemanual.pdf. Version 133. Make Noise.

[42] Novation. *Launchkey Mini [MK3]*. https://us.novationmusic.com/products/launchkey-mini-mk3. Accessed: 2023-12-06. 2023.

[43] V. Pujari. "Muscle Memory and the Brain: How Physical Skills are Stored and Retrieved". In: *Journal of Advanced Medical and Dental Sciences Research. https://doi. org/10.21276/jamds* (2019).

[44] S. Püst, L. Gieseke, and A. Brennecke. "Interaction Taxonomy for Sequencer-Based Music Performances". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Shanghai, China, June 2021. DOI: 10.21428/92fbeb44.0d5ab18d.

[45] H. Rabbits. *Orca: Live Coding Tool.* https://hundredrabbits.itch.io/orca. Accessed: 2023-11-22. 2023.

[46] L. Savoldy. *hsavoldy/GALiCA: Thesis Version.* Version v1.0.0. Dec. 2023. DOI: 10.5281/zenodo.10368850.

[47] T. Sayer. "Cognition and Improvisation: Some Implications for Live Coding". In: (July 2015). DOI: 10.5281/ZENODO.19328.

[48] TOPLAP. *All things live coding.* https://github.com/charlespwd/project-title.

[49] TOPLAP. *Original TOPLAP Draft Manifesto.* https://toplap.org/wiki/ManifestoDraft. Accessed: 2023-12-05.

[50] TSchenke. *File:Vestax VCI-380 MIDI DJ controller.* https://commons.wikimedia.org/w/index.php?curid=43863910. Accessed: 2023-12-12. Licensed under CC BY-SA 4.0. 2015.

[51] G. Wang, P. R. Cook, and S. Salazar. "ChucK: A Strongly-timed Computer Music Language". In: *Computer Music Journal* 39 (Dec. 2015). DOI: 10.1162/COMJ_a_00324.

[52] W. J. Wrigley and S. B. Emmerson. "The experience of the flow state in live music performance". In: *Psychology of Music* 41.3 (May 2013), pp. 292–305. ISSN: 0305-7356, 1741-3087. DOI: 10.1177/0305735611425903.